AUTOMATED DETECTION OF INJECTION VULNERABILITIES IN WEB APPLICATIONS

BY

SRUTHI BANDHAKAVI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor Marianne Winslett, Chair & Director of Research
Professor Madhusudan Parthasarathy
Professor Carl Gunter
Professor Samuel T. King
Professor William H. Winsborough, University of Texas at San Antonio

Abstract

Web applications and browsers embed code from different origins, each of which could have different levels of trustworthiness. These applications provide rich functionality to the users by interacting with each other and with the host machine—usually by sending and receiving information. The interaction of code and data from various sources in the web applications and browsers has resulted in the emergence of behaviors that could threaten the confidentiality, integrity, and availability properties of the infrastructure on which the web applications and web browsers run and that of the resources they handle. *Injection*, *cross-site scripting (XSS)*, *cross-site request forgery (XSRF)*, etc. are some examples of vulnerabilities that could allow malicious attacks on web applications and their infrastructure. Of the many attacks on web applications, injection attacks are one of the most popular ones. In this dissertation, we describe two different analysis techniques that could be used to automatically prevent and/or detect two specific types of injection vulnerabilities— *SQL injection attacks* on web applications and *cross-context scripting attacks* in Firefox browser extensions.

In case of vulnerabilities like SQL injection, malformed user input alters the SQL query issued. This query when processed allows the attacker to gain unauthorized access to the back-end server and database, and extract or modify sensitive information.

Like mobile phone apps, browser extensions are widely popular third-party-written weakly-vetted pieces of code. They expand the functionality of browsers by interposing on and interacting with browser-level events and data. In this thesis, we examine the cross-context scripting vulnerabilities occurring in the Firefox browser extensions. Firefox extensions are one of the main reasons for Firefox's popularity—there are thousands of third-party extensions available to the users. The extensions usually have the same privilege level as the browser. Cross-context scripting attacks are caused when some specially crafted inputs from the content pages (web pages, RSS feeds, *etc*) are executed in the privileged context of the

web browser, often by injecting malicious code into the extensions, which run with the same privileges as the web browser.

In both the above injection vulnerabilities, there is a legitimate flow of information from the untrusted sources to the trusted application. However, *the inputs from the untrusted sources force the trusted applications to behave in ways that are not intended by the application developers*. In case of SQL injection attacks, a malicious user of the web application inserts inputs that produce SQL queries with structures different from the query structures that the programmer intended. In case of cross-context scripting attacks, the content (either RSS feeds or web pages) contains malicious JavaScript, which when accessed by the browser extension changes its expected run-time behavior. This change in the expected behavior occurs if the programmers have written buggy code with inadequate sanitization or no sanitization at all of untrusted inputs.

To detect SQL injection attacks, apart from knowing that the SQL query is tainted by the inputs from the user, we also need to know exactly which parts of the query are tainted by the user inputs. A characteristic diagnostic feature of SQL injection attacks is that they change the intended structure of queries issued. Based on this insight, we propose a technique to detect and prevent SQL injection attacks. This technique dynamically mines the programmer-intended query structure on any input, and detects attacks by comparing the structure of the actual query issued with the programmer-intended query structure. We propose a simple and novel mechanism, called CANDID, for mining programmer-intended queries by dynamically evaluating runs over benign candidate inputs. We show that this mechanism is theoretically well founded and is based on inferring intended queries by considering the symbolic query computed on a program run. Our approach has been implemented in a tool called CANDID that retrofits web applications written in Java to defend them against SQL injection attacks. We report extensive experimental results that show that our approach performs remarkably well in practice.

To understand how to find cross-context scripting vulnerabilities, we study the Firefox browser extension architecture. In our analysis, we find that most of the vulnerabilities in the Firefox browser extensions can be modelled as information flows between different syntactic program elements in the extensions' JavaScript code. Based on this insight, we propose a new analysis framework called VEX, which uses static taint analysis of JavaScript programs to identify flow patterns that represent flows between untrusted sources and executable sinks. We de-

scribe several patterns of flows that can lead to privilege escalations in Firefox extensions. VEX analyzes Firefox extensions for such flow patterns using high-precision, context-sensitive, flow-sensitive static analysis. We subject 4303 browser extensions to the analysis, and VEX finds 5 of the 18 previously known vulnerabilities and 7 previously unknown vulnerabilities. Unlike previous works on browser extension security which use dynamic information flow analysis techniques for detecting vulnerabilities at run-time, VEX uses a static analysis technique. Consequently, VEX can be used to analyze thousands of extensions offline and imposes no performance overhead at run-time.

The main contribution of the thesis is to show that *static and dynamic information flow analysis techniques can be used effectively to detect programmers' intent and thereby detect and/or prevent script injection vulnerabilities.* We show that the application architecture, features of the programming language that is used to write the application, the complexity of the input language, and the characteristics of the vulnerabilities play a major role in designing useful information flow analysis techniques for detecting the vulnerability. We describe these considerations in this thesis. We also present an extensive comparison of our approaches with the related work in the area.

To my family and friends, for their love and support.

Acknowledgments

All through my Ph.D., I obtained guidance, advice, support and encouragement from all the wonderful people who were around me. These acknowledgements will not be enough to convey my gratitude to all the people involved, but I will try.

I will start by thanking my advisors, Professors Marianne Winslett and P. Madhusudan. I thank them for providing me so many opportunities to learn and grow over the course of my Ph.D. Professor Winslett provided me with all kinds of opportunities to grow as a researcher right from interacting with other researchers, reviewing papers, writing grant proposals, to mentoring undergraduates. Professor Madhusudan spent many hours with me discussing research and provided me with a wealth of information. I really appreciate the guidance they gave me in both professional and personal matters.

I also want to express my gratitude towards the members of my dissertation committee: Professors Carl Gunter, Samuel T. King, and William Winsborough. I thank Professor Samuel King for providing me with great feedback that guided me to think in interesting directions.

I also write these acknowledgments with a sad note on the untimely demise of Professor William Winsborough. Professor Winsborough was not just a guide, but a cherished friend. He was an enthusiastic researcher, who cared deeply about his students. I spent countless hours with him discussing research. I wish I could thank Professor Winsborough in person, but I hope he knew how much I appreciated his support during the formative years of my PhD. I will miss him dearly.

Donna Coleman and Elaine Wilson provided invaluable assistance with all of my administrative tasks. They always welcomed my questions with a smile and never failed to assist me. I would also like to thank the administrative staff: Mary Beth Kelly, Barbara Leisner, Rhonda McElroy, Lynette Lubben, Kathy Runck, Holly Bagwell, for answering my numerous questions over the years.

I am thankful to my co-authors, Prithvi Bisht, Sandro Etalle, Jerry Den Hertog, Bruno P. S. Roacha, V. N. Venkatakrishnan, for the several insightful dis-

cussions. Thanks are also due to my colleagues Sridhar Duggirala, Pranav Garg, Ragib Hasan, Rajesh Karmani, Adam Lee, Thanh Huong Luu, Kazuhiro Minami, Soumyadeb Mitra, Edgar Pek, Pavithra Prabhakar, Rishi Sinha, Naoki Tanaka, Arash Termehchy, Joana M. F. da Trindade and Charles Zhang for giving me the much needed feedback on my presentations. I had great discussions with my undergraduate interns: Shikhar Agarwal, Wyatt Pittman and Nandit Tiku, who worked hard to ensure that the VEX project was a success.

I thank my friends at UIUC, Nana Arizumi, Namrata Batra, Sarah Brown, Jay Chheda, Piyush Gupta, Revathi Jambunathan, Fariba Khan, Sibin Mohan, Namita Narain, Seema Kamath, Rajesh Karmani, Mohit Kumbhat, Veena Paliwal, Pavithra Prabhakar, Ramya Raviprakash, Vijay Reddy, Hardik Thakker, Mehul Thakkar, Utkarsh Sharma, Varashank Shukla, Komal Shukla, Anjali Sridhar, for providing the much needed encouragement, entertainment and nourishment. I also thank my worderful team at Tae Kwon Do at UIUC for supporting me and teaching me the skills that could be applied inside and outside the doe-chang (training room). The group of Women in Computer Science (WCS) provided me with a lot of information.

I am forever in debt to my mother, aunts, uncles and my extended family, who constantly encouraged me to achieve my dreams. I especially want to thank my sister, Sindhura Bandhakavi, for being the responsible one in the family. Last but not the least, I want to thank my husband, Pannagadatta Shivaswamy, for being with me through the highs and lows of the PhD process and being a pillar of strength throughout. I would not have survived the process without his support.

Table of Contents

Chapter 1 Introduction 1
1.1 Overview of Injection Vulnerabilities and Attacks 2
1.2 Detection and/or Prevention of Injection Vulnerabilities 5
1.3 Research Contributions 9
1.4 Thesis Organization 10

Chapter 2 Background 11
2.1 Detection and Prevention of Injection Flaws 11
2.1.1 Sanitization and Fuzz Testing 11
2.1.2 Static Analysis 12
2.1.3 Dynamic Analysis 12
2.1.4 Hybrid Approaches 12
2.2 Overview of the JavaScript Programming Language 12
2.2.1 JavaScript Features That Need To Be Analysed 16
2.3 Static Specification and Analysis of JavaScript 19
2.3.1 Applications of Analysis Techniques 20

Chapter 3 Detection and Prevention of SQL Injection Attacks 22
3.1 Overview of CANDID 23
3.1.1 An Example 23
3.1.2 Our Approach 25
3.2 Formal Analysis Using Symbolic Queries 27
3.2.1 SQL Injection Defined 29
3.3 The CANDID Transformation 34
3.3.1 Resilience of CANDID 36
3.4 Implementation and Evaluation 39
3.4.1 Transformation 40
3.4.2 Application Examples 40
3.4.3 Attack Suite 41
3.4.4 Experiment Setup 42
3.4.5 Attack Evaluation 43
3.4.6 Performance Evaluation 43
3.5 Related Work 45
3.5.1 Vulnerability Detection Using Static Analysis 46
3.5.2 Defensive Techniques that Prevent SQLCIA 46

Chapter 4 Detection of cross-context scripting vulnerabilities in Browser Extensions 51
4.1 Threat Model, Assumptions, and Usage Model 52
4.2 VEX Information Flow Patterns 53
4.2.1 Untrusted Sources 54
4.2.2 Executable Sinks 55
4.3 Static Information Flow Analysis of JavaScript 56
4.3.1 Core JavaScript Syntax 58
4.3.2 Abstract Heaps 60
4.3.3 Abstraction Function 61
4.3.4 Abstract Operational Semantics 62
4.3.5 Handling the Features of JavaScript 84
4.3.6 A Note on Soundness 87
4.4 Implementation and Evaluation 88
4.4.1 Evaluation Methodology 89
4.4.2 Experimental Results 89
4.4.3 Successful Attacks 93
4.4.4 Flows That Do Not Result in Attacks 97
4.5 Related Work 99
4.5.1 Firefox Browser Extension Security 99
4.5.2 Security of Extensions to Other Browsers 101
4.5.3 Operational Semantics of JavaScript 102
4.5.4 Comparison With Related Static Analyses of JavaScript . 102

Chapter 5 Conclusions 108
5.1 Conclusions 108
5.2 Future Research Directions 112
5.2.1 Generating Attack Inputs 112
5.2.2 Securing Extensible Software 113

References 116

Chapter 1

Introduction

With the advent of Web 2.0 and the cloud computing paradigm, Web browsers have evolved from a platform to display static data to one facilitating client-side processing and more complicated Web applications. Increasingly, users are relying on Web applications to perform tasks like shopping, business, banking, healthcare, social networking, and other critical interactions that involve exchange of sensitive information. Different organizations in turn are relying on providing more and more services through the Web to increase the availability of their offerings, and to cut costs.

Such Web applications embed code and data obtained from different sources, each of which could have different levels of trustworthiness. In fact, entire revenue models of some companies are based on allowing third-party generated content to be displayed on the pages generated by their Web applications. For example, Web services like Google and Facebook allow advertisements to be displayed in the Web pages they generate. Facebook allows third-party developers to create and upload new applications. Web applications can also create mashups, which are Web pages created by combining data, functionality and presentation from two or more existing services. An apartment search mashup for a particular location can be created by combining content from Web sites like Craigslist and displaying them on the Google maps obtained for that location.

To accomodate the growing needs of the Web application users, Web browsers are allowing users to customize their browsing experience by installing new applications (called *browser extensions*). Browser extensions are third-party developed applications that extend the functionality of the browser and often run with full privileges of the browser.

The interaction of code and data from various sources in the Web applications and browsers has resulted in the emergence of behaviors that could threaten the confidentiality, integrity, and availability properties of the infrastructure on which the Web applications and Web browsers run. *Injection*, *cross-site scripting (XSS)*,

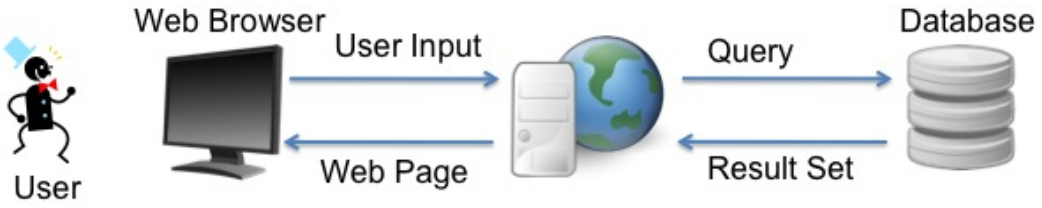

Figure 1.1: Architecture of a typical Web application

cross-site request forgery (XSRF), etc. are some examples of vulnerabilities that could allow malicious attacks on Web applications and their infrastructure. Descriptions of these vulnerabilities can be found in the list of of top ten Web application security vulnerabilities published by Open Web Application Security Project (OWASP) in 2010 [1]. Among all the vulnerabilities listed, OWASP ranks *injection vulnerabilities* as the most prevalent.

The OWASP list describes injection vulnerabilities as follows: *Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.* This thesis describes the different analysis techniques we have developed for detecting and preventing two kinds of injection attacks, namely SQL injection attacks on Web applications and cross-context script injection attacks on Firefox Web browser extensions.

1.1 Overview of Injection Vulnerabilities and Attacks

SQL injection: SQL injection attacks occur when the inputs to Web applications are used to attack the back-end database layers of the Web servers. A typical Web application architecture is shown in Figure 1.1. The Web applications generate Web pages that are displayed on the Web browser. These applications provide rich functionality to users by interacting with each other and with the host machine – usually by sending and receiving information. Most Web applications also allow their users to input information, which determines the control flow and the output of the Web application. A Web server uses this information to create a query to send to the back-end database. The database generates the results and returns them to the Web application, which then displays the results in the user's

Web browser. A malicious user of the Web application can manipulate the inputs so that the application produces malicious queries, thereby generating SQL injection attacks.

We can illustrate SQL injection using an example. Consider a simple online phone book manager that allows users to view or modify their phone book entries. Phone book entries are private, and are protected by passwords. To view an entry, the user enters his user name and password in a HTML form provided by the Web application. The inputs from the HTML form are directly supplied to a procedure that generates a SQL query. Hence, if the user supplies the string "`John`" for the username and "`correct-password`" for password, the following query will be generated:

```
SELECT * from phonebook WHERE username='John' AND
            password='correct-password'
```

To attack the Web application, a malicious user can supply the string "`John' OR 1=1 --`" for the username, and "`not-needed`" for the password as inputs, which will make the program issue the SQL query:

```
SELECT * from phonebook WHERE username='John' OR 1=1
              --'password='not-needed'
```

The query selection clause contains the tautology 1=1, and given the injected OR operator, the `SELECT` condition always evaluates to true. The sub-string "`--`" gets interpreted as the comment operator in SQL, and hence the portion of the query that checks the password gets commented out. The net result of executing this query is that the malicious user can now view all the phone book entries of all users. Using similar attack queries, the attacker can construct attacks that delete phone number entries or modify existing entries with spurious values. A program vulnerable to an SQL injection attack is often exploitable further, as once an attacker takes control of the database, he can often exploit it (for example using command-shell scripts in stored procedures in the SQL server) to further violate the confidentiality and integrity of the machine.

Cross-context scripting: Cross-context scripting [2] vulnerabilities are subtle injection vulnerabilities, that expose the browser's user to a disastrous attack from the Web, often just by viewing a Web page. These attacks violate the privilege separation between the Web browser and the content displayed in the browser.

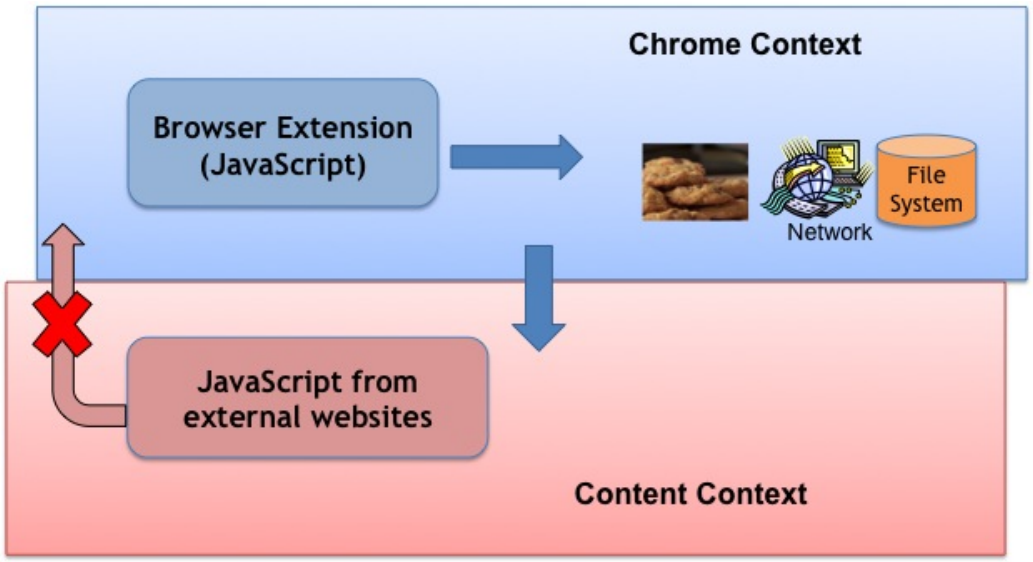

Figure 1.2: Architecture of the Firefox browser

One instance of cross-context scripting vulnerabilities occurs in the Firefox Web browser. Similar issues could arise in other extensible browsers like Chrome, Safari, and Internet Explorer.

The reason for such vulnerabilities is evident when we understand the Firefox browser architecture as shown in Figure 1.2. Firefox has two privilege levels: *content* for the Web page displayed in the browser's content pane, and *chrome* for elements belonging to Firefox. The Firefox browser also allows the user to download third-party written software called *browser extensions*, which help meet the varied needs of the broad user population. Browser extensions expand the functionality of browsers by interposing on and interacting with browser-level events and data. Firefox code and extensions run with full chrome privileges, which enables them to access all browser states and events, OS resources like the file system and network, and all Web pages. Extensions also can include their own user interface components via a *chrome document*, which can run with full chrome privileges. Content privileges are more restrictive than chrome privileges. For example, a page loaded from `illinois.edu` can only access content from `illinois.edu`.

One of the causes for cross-context scripting vulnerabilities is the interaction between extensions in the chrome context and the content, which is facilitated by the APIs provided by Firefox to communicate across protection domains. As the Mozilla developer site explains, "One of the most common security issues with

extensions is execution of remote code in privileged context. A typical example is an RSS reader extension that would take the content of the RSS feed (HTML code), format it nicely and insert into the extension window. The issue that is commonly overlooked here is that the RSS feed could contain some malicious JavaScript code and it would then execute with the privileges of the extension – meaning that it would get full access to the browser (cookies, history etc) and to user's files" [sic].

There are thousands of JavaScript browser extensions for Firefox. The extensions could be written by in-experienced programmers, who often do not know how to secure their code or how to properly sanitize the inputs. When coupled with the dynamic nature of JavaScript semantics, which allow certain program constructs to generate code during the execution of the program, this may give rise to vulnerabilities. These vulnerabilities can be exploited by malicious Web content developers to steal confidential information like bank account details, passwords, cookies, etc. They may also inject code that re-directs a user to a malicious Websites. Currently, the extensions are vetted by manual inspection, which does not scale well and is subject to human error.

1.2 Detection and/or Prevention of Injection Vulnerabilities

In both the above injection vulnerabilities, there is a legitimate flow of information from the untrusted sources to the trusted application. *The inputs from the untrusted sources result in behaviors that are not intended by the application developers*. In case of SQL injection attacks, a malicious user of the Web application inserts inputs that produce SQL queries with structures different from the query structures that the programmer intended. In case of cross-context scripting attacks, the content (either RSS feeds or Web pages) contains malicious JavaScript, which when accessed by the browser extension changes its expected run-time behavior. This change in the expected behavior occurs if the programmers have written buggy code with inadequate sanitization or no sanitization at all of untrusted inputs.

Therefore, in both SQL injection and cross-context scripting, if one can understand the programmers' intent, any deviation from the intent can be considered an attack. This thesis describes two analysis techniques, that use information flow analysis, for understanding programmer intent and thereby facilitating automated

detection and/or prevention of such injection attacks.

SQL Injection: The problem of SQL injection is one of information flow integrity [3, 4]. The semantic notion of data integrity requires that untrusted input sources (i.e., user inputs) must not affect trusted outputs (i.e., structure of SQL queries constructed). Notions of *explicit information flows* that track a relaxed version of the above data integrity problem are suitable for handling this problem.

Research on SQL injection can be broadly classified into two basic categories: *vulnerability identification* approaches and *attack prevention* approaches. The former category consists of techniques that identify vulnerable locations in a Web application that may lead to SQL injection attacks.

In order to avoid SQL injection attacks, a programmer often subjects all inputs to input validation and filtering routines that either detect attempts to inject SQL commands or render the input benign [5, 6]. The techniques presented in [7, 8] represent the prominent static analysis techniques for vulnerability identification, where code is analyzed to ensure that every piece of input is subject to an input validation check before being incorporated into a query (blocks of code that validate input are manually annotated by the user). Solutions based on tracking of explicit information flows have been implemented by mechanisms such as tainting [9, 10, 11] and bracketing [12].

While these static analysis approaches scale well and detect vulnerabilities, their use in addressing the SQL injection problem is *limited* to merely identifying potential unvalidated inputs. The tools do not provide any way to check the *correctness* of the input validation routines, and programs using incomplete input validation routines may indeed pass these checks and still be vulnerable to injection attacks. Although input validation routines can serve as a first level of defense, it is widely agreed [13] that they cannot defend against sophisticated attack techniques (for instance, those that use alternate encodings, and database commands to dynamically construct strings) to inject malicious inputs to SQL queries.

A much more satisfactory treatment of the problem is provided by the class of attack prevention techniques that *retrofit* programs to shield themselves against SQL injection attacks [14, 15, 10, 9, 11, 16, 17, 12]. These techniques often require little manual annotation, and instead of detecting vulnerabilities in programs, offer preventive mechanisms that solve for the programmer the problem of defending the code against injection attacks.

A more fundamental technique to the problem of defending SQL injection comes from the commercial database world, in the form of `PREPARE` statements. These statements allow a programmer to declare (and finalize) the structure of every SQL query in the application. Once issued, these statements do not allow malformed inputs to further influence the SQL query structure, thereby avoiding SQL vulnerabilities altogether. This is in fact a *robust* mechanism to prevent SQL injection attacks. However, retrofitting an application to make use of `PREPARE` statements requires manual effort in specifying the intended query at every query point, and the effort required is proportional to the complexity of the Web application.

The above discussion raises a natural question: Could we *automatically infer* the structure of the programmer-intended query structures at every query issue point in the application? A positive answer to this question will address the retrofitting problem, thereby providing a robust defense for SQL injection attacks.

Based on the insight that the *the attack inputs change the intended structure of queries issued*, we propose a technique called CANDID to detect and prevent SQL injection attacks [18]. CANDID detects SQL injection attacks by dynamically inferring the programmer-intended query structure on any input, and comparing it against the structure of the actual query issued. CANDID's dynamic analysis technique is based on the fact that if we can extract a slice of the program statements that are used to generate the user query, and execute these statements on benign inputs, then we will get a benign query. If the structure of this benign query is different from the query generated using the user inputs, then we can infer that the user inputs changed the query structure and therefore they are malicious. In CANDID, we simulate the process of extracting the program slice by instrumenting the program such that it computes both the user and the benign queries simultaneously.

Cross-context Scripting: In the case of browser extensions, most cross-site scripting attacks involve the attacker injecting JavaScript into a data item that is subsequently executed by the extension under full browser privileges, thereby *changing the extension's program structure*. The vulnerabilities are caused by some special commands provided by the JavaScript programming language, which execute the inputs obtained from the untrusted content. Therefore, our key insight is that *extension vulnerabilities often translate into explicit information flows from injectable sources to executable sinks*.

Several *dynamic* analysis techniques with static instrumentation have been proposed for JavaScript to check information-flow properties [19, 20]. *SABRE* [21] is a framework for dynamically tracking in-browser information flows for analyzing JavaScript-based browser extensions. The taints are tracked by modifying the JavaScript interpreter. In contrast, Djeric *et al.* [22] dynamically track taints in both the browser's native code and the script interpreter. Ter Louw *et al.* [23] highlight some of the potential security risks posed by browser extensions, and propose run time support for restricting the interactions between browsers and extensions.

Although dynamic techniques are useful in preventing certain types of script injection attacks if they are enforced by the Web browser, they suffer from a few drawbacks. When a questionable flow is detected dynamically, the browser has to either choose an appropriate action (which might be overly restrictive) or ask the user to choose an action (which might lead to an attack if the user chooses a wrong option). Additionally, dynamic techniques impose a performance and memory overhead on the browser because of the need to keep track of the security label for every JavaScript object inside the browser. One of our main motivations was to overcome these drawbacks while facilitating an analysis that scales to thousands of extensions.

More recently, researchers have developed static information flow analysis methods for JavaScript [24, 25]. Various language features of JavaScript make precise static analysis hard. JavaScript is a dynamic programming language – new code can be generated and executed during run-time using methods like `eval`, `settimeout`, etc. Additionally, JavaScript supports higher order functions – functions can be assigned to program variables and passed parameters to other functions. Apart from having a reasonable way to handle all these non-standard features, the extensions access the browser API and the DOM API, which also should be included in the analysis of programs.

In [24] the authors essentially perform a *context-insensitive* and *flow-insensitive* static analysis on the code, and delegate analysis of dynamic code to runtime checks. Guarnieri *et al.* [25] propose a mostly-static enforcement for JavaScript analysis, which is *context-sensitive* but *flow-insensitive*.

To automatically detect vulnerable extensions, we propose VEX, a novel high-precision static-information flow analysis technique for analyzing JavaScript source code [26]. We identify key information flow patterns (untrusted source and executable sink pairs) that can lead to security vulnerabilities, and the VEX static

analysis checks extensions for the presence of such flows.

1.3 Research Contributions

My thesis shows that *information flow analysis techniques, both static and dynamic, can be used to detect script injection vulnerabilities. By trading soundness for tractability, static analysis techniques can be effective for bug detection in complex applications.*

Detection and Prevention of SQL Injection Attacks: Our work on SQL injection makes the following contributions:

- We propose a simple and novel mechanism, called CANDID, for dynamically generating programmer intended queries for each query generated from the user inputs.
- We show that there is a formal basis for this dynamic approach using the notion of symbolic queries.
- We develop a fully automated, program transformation mechanism for Java programs that employs this technique, with a discussion of practical issues and resilience to various artifacts of Web applications.
- We perform a comprehensive evaluation of the effectiveness of CANDID's attack detection and its performance overheads.

Detection of Cross-Context Scripting Attacks: Our work on cross-context scripting makes the following contributions:

- We describe several vulnerable patterns of flows as well as unsafe programming practices that could lead to cross-context scripting attacks in Firefox extensions.
- We enumerate the features of JavaScript programming language that make the programs hard to analyze statically.
- We develop VEX, a framework for analyzing Firefox browser extensions for vulnerable patterns. VEX contains a high-precision flow-sensitive, and context-sensitive static analysis as its core.

- We subject thousands of browser extensions to the VEX analysis. VEX finds a dozen exploitable vulnerabilities, seven of which were previously unknown. VEX also finds hundreds of bad programming practices.
- We present a comprehensive analysis of the different static analyses for JavaScript and compare them with VEX.

1.4 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 gives the background of different approaches to detect and prevent injection attacks. It also gives an overview of the different features of JavaScript and then describes the different static analysis techniques defeloped for JavaScript and what these techniques are used for. Chapter 3 describes our approach for detecting and preventing SQL injection attacks. Chapter 4 summarizes our approach on detecting cross-context scripting vulnerabilities. We present our conclusions and directions for future work in Chapter 5.

Chapter 2

Background

In this chapter, we provide an overview of different kinds of ways in which injection attacks could be detected and prevented.

2.1 Detection and Prevention of Injection Flaws

2.1.1 Sanitization and Fuzz Testing

The most basic form of preventing injection attacks is through sanitization of inputs to the web application. Web applications can sanitize inputs by writing custom sanitization functions. They usually use regular expressions [27] to ensure that the input strings do not have malicious scripts embedded in them. However, it is hard to write correct sanitization functions that take care of all the corner cases. They could be either too restrictive and generate a lot of false positives or worse they could have missed some cases [6] like encodings in a different language, *etc*, thereby allowing malicious inputs to be executed. Therefore, sanitization routines should not be the only mechanism to prevent attacks on applications. Another way to perform sanitization is to use a different encoding for the trusted strings defined in the Web application. Any script in the input would have a different encoding than the script in the Web application and therefore they can be separated.

Web application developers generally use fuzzers to discover security vulnerabilities in their applications before deployment. Fuzzers are tools that generate random input strings from a given language specification, which can be used to test if the applications filter those strings. The random strings can be generated using a collection of attack input specifications. A comprehensive list of cross-site scripting attack vectors is given in [28]. Several fuzz testing tools [29, 30, 31, 32, 33, 34, 35] have been developed for detection of SQL injection attacks and cross-site scripting (XSS) attacks.

2.1.2 Static Analysis

Several static analysis techniques have been proposed for detecting and preventing injection attacks. The main idea is to find the taints from untrusted sources to trusted sinks and to check if that particular path can lead to a vulnerability.

2.1.3 Dynamic Analysis

There are several different kinds of dynamic analysis proposed for detecting and preventing injection attacks. One technique is to instrument the applications statically to keep track of the taints dynamically. Several *dynamic* taint analysis techniques with static instrumentation have been proposed for different programming languages to check information-flow properties.

2.1.4 Hybrid Approaches

In these approaches, the analysis statically infers a property that needs to be satisfied and instruments the application to ensure that such a property is satisfied at run-time. An example would be an analysis technique where static string analysis is performed over the Web applications to infer the model of the query string. Any query generated dynamically should confirm to this model.

2.2 Overview of the JavaScript Programming Language

JavaScript is a prototype-based, object-oriented scripting language. The language has both imperative and functional features. It is a weakly typed language and allows dynamic code generation. In this section we will give an overview of the various language features of JavaScript. Some of the code samples are taken from the book "JavaScript: The Definitive Guide" [36] and the IBM JavaScript Security Test Suite [37]. In Section 2.2.1, we point out the features of JavaScript, which need to be handled effectively by any analysis of JavaScript.

- **Objects:** A JavaScript object represents a collection of (name, value) pairs. The names given to the values are called object properties. The values could

be primitive, like numbers and strings, or they could be composite values like other objects (even functions). Object properties can be accessed either by the field access notation (*obj.prop*) or by using an associative array notation (*obj["prop"]*). This means that an object property can dynamically instantiate, as shown in the following code, where the function parameter "b" is instantiated dynamically to the string "URL" when the function "foo" is called.

```
function foo(a, b) {
  var t = a[b];
  document.write(t);
}
foo(document, "URL");
```

JavaScript objects can be dynamically modified i.e. new properties can be added, old properties can be associated with new values, *etc* during program execution. Therefore, the object signature need not be constant during the execution of the program.

- **Arrays:** Arrays are objects that contain numbers as property names and therefore contain an ordered collection of numbered values. Array indices should be between 0 and $2^{32} - 1$. If the array index is too large, a negative number, or a floating-point number, it is converted to a string and used as a property name. An array object has a few built-in library functions associated with it for array processing. Some examples of the built in methods are *join, reverse, sort, concat, slice, splice*, *etc*. An array can hold values of different data types, making it different from the arrays in other programming languages.

- **Function:** Functions are also JavaScript objects, which can be defined once and can be invoked, or executed, or used as constructors any number of times. JavaScript functions can be invoked using a variable number of arguments. These arguments are stored in an array property named *"arguments"*. Values of any type can be passed to the function as arguments, since JavaScript is a loosely typed programming language. JavaScript also allows declaration of nested functions.

  JavaScript supports higher-order functions similar to functional programming languages. The functions are treated as first-class data values. They

can be assigned to variables, passed as parameters to other functions, stored as properties in objects or in array variables, *etc*.

- **Execution Context:** When the JavaScript interpreter starts up, it creates a *global object* to hold the various global properties defined in the JavaScript program. This global object is then initialized with the JavaScript core objects like *String, Array, Object, Function, Date, etc* . Any code that is not part of a function runs with the global object as its execution context, i.e. local variables are defined in the global object.

  Whenever a new function begins to execute, a new object (called the *call object*) is created to act as an execution context. While the body of a function is executing, the function arguments and local variables are stored as properties of this call object.

- **Variable Scoping:** The `var` keyword is used to declare the variables. Variables can also be implicitly declared without the `var` keyword, in which case they are considered to be global variables even if they are declared inside a function body.

  JavaScript has function-level scope. All variables declared in a function, no matter where they are declared, are defined throughout the function.

  Every execution context in JavaScript is associated with a scope chain. Whenever there is a need for an variable lookup, the current scope object is checked to see if the variable is in that scope. If not, the scope object's scope is checked for the presence of the variable. A function declared in the global scope has the function's call object as its current scope and the global scope in its scope chain.

- **Function Scoping:** Functions in JavaScript are lexically rather than dynamically scoped. This means that they run in the scope in which they are defined, not the scope from which they are executed. When a function is defined, the current scope chain is saved and becomes part of the internal state of the function.

- `this` **property:** The `this` property refers to the calling context in which a particular method is called. The semantics of this property can be explained using the following figure.

```
1  var a1 = "in global";
2  var obj = {a1:"in obj"};
3  function foo(){
4    var a1 = "in foo";
5
6    function bar(){
7          var id1 = a1;
8          var id2 = this.a1;
9          document.write("id1: " + id1 + " \t\t\t\t
               id2:" + id2);
10   };
11   bar(); // id1 = "in foo" , id2 = "in global"
12
13   obj.test = bar;
14   obj.test(); // id1 = "in foo", id2 = "in obj"
15 }
16 foo();
```

In the above example, a function "foo" is defined to have a local variable "a1". There is global variable which shares the same identifier "a1" and another global variable which references an object with a property named "a1". The function "bar" has two accesses to the identifier "a1", "id1" and "id2" which are assigned "a1" and "this.a1" respectively. During the program's execution, when the "bar" method is called as a function,"id1" and "id2" get values "in foo" and "in global" respectively. However, when the same bar function is assigned to an object property and called as a method of the object "obj", "id1" and "id2" get values "in foo" and "in obj" respectively. The "this" property refers to the calling context of the method "test", which is the object "obj".

- **Prototype-based Inheritance:** JavaScript uses prototype-based inheritance [38]. Every object in the JavaScript heap has a special `@Proto` property, which is used to specify inheritance chains. Additionally, every function (that can be used as a constructor in `new`) has a `prototype` property. This `prototype` property is used to instantiate the `@Proto` property when a new object is created using the function constructor. An object inherits all the properties of its `@Proto` and of all the objects in the prototype's `@Proto`

chain.

- **Dynamic Code Generation:** The core JavaScript language contains special methods (like `eval`, `settimeout`, `Function`, `setinterval`, *etc*) that accept a string value as a parameter. At run-time, this string value is parsed and executed at the program point where the methods are called. These methods can be used to generate code dynamically based on the inputs given to the program.

2.2.1 JavaScript Features That Need To Be Analysed

In addition to the non-standard scoping rules and prototype-based inheritance, any analysis engine should be able to handle the following features in order to effectively analyse the information flows in JavaScript code.

- **Object Accessed Using Associative Array Notation:** A property name can be computed dynamically using string operations as shown in the following code sample. The operation "a+b" results in the computation of the string "foo". "o["foo"]" is the same as accessing the property using the field access notation"o.foo".

```
var o = new Object();
o.foo = document.URL;
var a = "f";
var b = "oo";
document.write(o[a+b]);
```

  An analysis needs to have string handling capabilities to correctly handle array accesses like in the above example.

- **Aliasing:** In JavaScript, every construct is an object. The JavaScript objects can be accessed using their variable references. These variables can also be assigned to other variables, thereby creating an alias for the object. Aliasing combined with dynamic access and modification of objects and functions could result in unexpected vulnerabilities. The following code show the example of the code where the "eval" function is aliased to a program variable.

```
1 e = eval;
2 e(window.content.document.getElementById(''Form1''));
```

An analysis algorithm has to ensure that it keeps track of this kind of aliasing. An analysis needs to keep track of the fact that "e" and "eval" are aliased in order to detect that there is a flow from "window.content.document" to "eval".

The following example shows an example of interprocedural aliasing – the object "p1.f.g.t" refers to the variable "a", which in turn points to the "window.location" object.

```
1  function foo(p1, p2) {
2    p1.f = new Object();
3    p1.f.g = new Object();
4    p1.f.g.t = p2;
5  }
6
7  var a = window.location.toString();
8  var b = new Object();
9  foo(b,a);
10 document.write(b.f.g.t);
```

- **Context-sensitive Function Call Handling:** JavaScript functions can be called from different context with different parameters. In the following code, there is a flow from "window.content.document" to "eval" when the function "foo" is called in line 4. However, there is no such flow when "foo" is called in line 5. An analysis has to differentiate between these two function calls.

```
1  function foo(x) {
2   eval(x)
3  }
4  foo(window.content.document);
5  foo(''var x = 10'');
```

- **Functions that take variable number of arguments:** Functions in JavaScript can take variable number of arguments. Any analysis has to take this into consideration when function summaries are being generated.

- **Handling of higher-order functions:** In JavaScript, the higher-order functions are used in several ways –as event handlers (which take function references as callbacks), as closures (functions that preserve the local variables

defined), *etc*. One example of such functions is the "settimeout" function, which takes a function reference or a function definition as its first argument. This functions is called after a certain timeout. In the following example, the "alertMsg" function is called with an argument "b" after 3 seconds. An analysis algorithm has to ensure that it also detects the function call in the "settimeout" function.

```
1  function timeMsg()
2  {
3      b = "var x = 10; alert(x)";
4      var t=settimeout("alertMsg(b)",3000);
5  }
6  function alertMsg(b)
7  {
8      eval(b);
9  }
10 timeMsg();
```

- **Core Library Functions and External JavaScript API:** JavaScript is a dynamic language. It provides powerful language features through its core library and also by allowing external libraries to be easily accessed and used. The HTML DOM API is one of the most common external libraries included with JavaScript. This API provides functionality to access and modify HTML content of a web page. Some examples of DOM methods are getElementById, innerHTML, appendChild, *etc*. The Firefox browser provides additional API (called the Component API), which allow the extension JavaScript programs to access the privileged contexts like the file system, network, cookies in the browser, *etc*. There are several external libraries available for JavaScript and they could also be included in the JavaScript program. Some common external libraries are JQuery, DoJo, Prototype, *etc*. An effective analysis should be able to handle all the new methods and features added by the external libraries.

  XMLHttpRequest object is an example of an external object associated with the DOM API. This object can be used by JavaScript programs to programmatically connect to their originating server via HTTP. The XMLHttpRequest object is associated with an event handler function which is called when the request sent to the external web page sends a response. In order

to capture the information flows in the program, the analysis has to capture the flow of information between the response and the rest of the program.

- **Manual and built-in sanitizers:** JavaScript also has some built in DOM API which could be used to sanitize certain content. For example, the `encodeURI` function can be used to encode the special character in a URL. The programmers can also introduce their own sanitizers manually. An analysis engine could have mechanisms to detect such sanitizers. However, since JavaScript allows dyanamic modifications of functions, the analysis should ensure that the sanitizers have not be overwritten in the code as shown in the following example.

```
1 var t = document.URL;
2 function k() {
3     encodeURI = function (str) { return str; }
4 }
5 k();
6 var u = encodeURI(t);
7 document.write(u);
```

2.3 Static Specification and Analysis of JavaScript

Operational Semantics. Yu *et al* [19] propose an operational semantics for a core subset of JavaScript, which allows the analysis to instrument the code with certain new operations. These new operations add a policy checking framework to the JavaScript program. Maffeis *et al* [39] defined a small-step operational semantics for the ECMAScript standard language corresponding to JavaScript, using which they analyze security properties of web applications. These operational semantics are one of the most comprehensive semantics covering all the features specified in the language. The authors also present a soundness proof for the semantics. Guha *et al* [38] propose an alternate operational semantics. In their semantics, they reduce JavaScript to a core calculus structured as a small-step operational semantics. They then present a de-sugaring process using which JavaScript programs can be converted into the programs written in the core calculus. They demonstrate the correctness of the semantics by running several real-world test suites using the

semantics and showing that they produce the same results as those produced by the full JavaScript interpreters. Taly *et al* [40] propose an operational semantics for a slightly restricted and modified subset of the JavaScript strict mode (ES5S) in the 5th edition of the ECMAScript Standard (ES5). The strict mode supports lexical scoping and closure-based encapsulation. The authors enforce a few more restrictions on the language to ensure that it is amenable to static analysis.

Type Systems. Several static and dynamic type checking and inference systems [41, 42, 43, 44, 45, 46, 47] were proposed for JavaScript to detect errors and vulnerabilities caused due to the lack of typing in JavaScript. Weber [46] lists out the different mechanisms proposed for type checking and type inferencing in JavaScript programs.

Points-to Analysis. Point-to analysis gives a picture of the different reference relations between the objects in the JavaScript program. Several [25, 48, 40] points-to analysis approaches have been proposed for different subsets of JavaScript.

Flow Analysis. Guha *et al* [49] build a static-flow analysis tool for the client portion of the Ajax web applications. Using the flow analysis, they build a flow graph of URLs (request graph) that the client-side program can invoke. Chugh *et al* [24] Propose an information-flow based approach for inferring the effects that a piece of JavaScript has on the website in order to ensure that key security properties are not violated. Bandhakavi *et al* [26] propose an data-flow analysis technique to detect flows between various program variables in the JavaScript program.

2.3.1 Applications of Analysis Techniques

- Comparing different JavaScript interpreters: The ECMA specification of JavaScript is a 200 page long text. Several JavaScript interpreters produce different results for different features of JavaScript based on their understanding of the specification. Specifying precise operational semantics can help in understanding these differences by comparing their semantics for different features. In this context, Maffeis *et al* [39] performed number of experiments with different implementations of JavaScript.

- Sandboxing client-side code: Several browser level sandboxes have been proposed to ensure that the client side JavaScript does not access and modify core JavaScript objects at run-time. Some examples of sandboxes are Caja, BrowserShield, browser instrumentation(CoreScript), ADSafe, FacebookJS *etc*. Guha *et al* [38] use their semantics to develop a safe sublanguage of JavaScript.

- Verifying correctness of sandboxes: Maffeis *et al* [50, 51, 52, 53] use their operational semantics for generating safe subsets of JavaScript and to manually prove that the so-called safe subsets of JavaScript are in fact vulnerable to certain attacks. Politz *et al* [41] propose a type System to encode and verify sand-boxing properties of the browser.

- Preventing run-time errors: Type system papers JavaScript is dynamic, weakly-typed (object but no classes) and has prototype based inheritance. It also allows dynamic addition of members to objects. These features could lead to run-time errors like buffer overflows, wrong type conversions, invocation of undefined values, *etc*. Several type based approaches have been proposed for detecting and preventing such errors.

- Detecting and preventing client-side vulnerabilities: Points-to analysis and flow analysis techniques [24, 25, 48, 49] have been used for detecting and preventing several client-side vulnerabilities like cross-site scripting, cross-site request forgery, modification of AJAX behavior, etc.

- Detecting browser extension vulnerabilities: Flow analysis techniques can also be used to find malicious information flow patterns in browser extension JavaScript code [54, 26].

Chapter 3

Detection and Prevention of SQL Injection Attacks

SQL injection attacks are one of the topmost threats for applications written for the Web. These attacks are launched through specially crafted user input on web applications that use low level string operations to construct SQL queries.

In this chapter we offer a solution for detecting and preventing SQL injection attacks using a technique that we call *dynamic candidate evaluation*. This technique automatically (and dynamically) mines programmer-intended query structures at each SQL query location, thus providing a robust solution to the retrofitting problem. Central to our approach are two simple but powerful ideas: (1) the notion that the symbolic query computed on a program path captures the intention of the programmer, and (2) a simple dynamic technique to mine these programmer-intended query structures using candidate evaluations.

Based on these ideas, we build a tool called CANDID (CANdidate evaluation for Discovering Intent Dynamically). CANDID retrofits web applications written in Java through a program transformation. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks. We support this claim by evaluating the efficacy of the tool in preventing a large set of attacks on a class of real-world examples.

The chapter is organized as follows: In Section 3.1 we informally present our approach using an example. We present the formal basis for the idea of deriving intended query structures in Section 3.2. Section 3.3 presents the program transformation techniques used to compute programmer-intended query structures. Section 3.4.1 presents the functional and performance evaluation of our approach through experiments on our tool CANDID. A much more detailed comparison with the related work is given in Section 3.5.

3.1 Overview of CANDID

3.1.1 An Example

Figure 3.1: An HTML Form Generated by the Web Application

```
1  void process-form(String uname, String pwd, boolean
     modify, String phonenum) {
2   if (modify == false){ /* Path 1. only display */
3    query = "SELECT * FROM phonebook WHERE username = '"
       + uname + "' AND password = ' " + pwd + "'";
4   }
5   else{  /* modify telephone number */
6     if (phonenum == " ") /* Path 2. delete entry */
7       query = "DELETE FROM phonebook WHERE username='"
          + uname + "' AND password = '" + pwd + " ' ";
8     else                 /* Path 3. update entry */
9       query = "UPDATE phonebook SET phonenumber =" +
          phonenum + "WHERE username = ' "  + uname + "'
           AND password = '" + pwd + "'";
10  }
11  sql.execute(query);
12 }
```

Figure 3.2: Phonebook Record Manager Application Code

To illustrate SQL injection, let us consider the web application for a simple online phone book manager, which generates an HTML form as depicted in Figure 3.1. This application allows users to view or modify their phone book entries,

which are private, and are protected by passwords. To view an entry, the user fills in her user name, and chooses the `Display` button. To modify an entry, she chooses the `Modify` button, and enters a different phone number that will be updated in her record. If this entry is left blank, and the modify option is chosen, her entry is deleted. The program that processes the input supplied by the form is also shown in Figure 3.2. Depending on the display/modify check-box and depending on whether the phone number is supplied or not, the application issues a `SELECT`, `UPDATE`, or `DELETE` query.

The inputs from the HTML form are directly supplied to procedure `process-form`, and hence the application is vulnerable to SQL injection attacks. In particular, a malicious user can supply the string "`John' OR 1=1 --`" for the username, and "`not-needed`" for the password as inputs, and check the display option, which will make the program issue the SQL query at line 3 (along the true branch of the conditional in line 2):

```
SELECT * FROM phonebook WHERE username='John' OR 1=1 --
    -' AND password='not-needed'
```

This contains the tautology 1=1, and given the injected OR operator, the `SELECT` condition always evaluates to true. The sub-string "`--`" gets interpreted as the comment operator in SQL, and hence the portion of the query that checks the password gets commented out. The structure of the original query that contained the "AND" operator is now changed to a query that contained an "OR" operator that uses a tautology. The net result of executing this query is that the malicious user can now view all the phone book entries of all users. Using similar attack queries, the attacker can construct attacks that delete phone number entries or modify existing entries with spurious values. A program vulnerable to an SQL injection attack is often exploitable further – once an attacker takes control of the database, he can often exploit it (for example using command-shell scripts in stored procedures in the SQL server) to gain additional access.

In order for an attack to be successful, the attacker must provide input that will ultimately affect the construction of a SQL query statement in the program. In the above example, the user name "`John' OR 1=1 --' `" is an attack input, whereas the input name "`John`" is not. An important observation that is used in SQL `PREPARE` statements, and also in the work [17, 12] is that a successful attack always changes the structure of the SQL query intended by the programmer of the

application. For the example given above, the attack input "`John' OR 1=1 --'`" results in a query structure whose condition consists of an "`OR`" clause, whereas the corresponding query generated using non-attack input "`John`" has a corresponding "`AND`" clause. Detecting change in query structure that departs from the one intended by the programmer is therefore a robust and uniform mechanism to detect injection attacks.

The problem then is to learn the structure of programmer-intended `PREPARE` query structures for various query issuing locations in the program. If this can be accomplished, then during program execution, the syntactic structures of the programmer-intended query and the actual query can be compared in order to detect attacks.

Several options are available to learn programmer intended queries. One approach is to construct valid query structures from safe test inputs [15]. The problem with a purely testing-based strategy is that it may miss some control paths in the program and may not be exhaustive, leading to rejection of valid inputs when the application is deployed. Another possibility is to use *static analysis* techniques [14] to construct the programmer-intended queries for each program point that issues a query. The effectiveness of static analysis is dependent on the precision of the string analysis routines. As we show the related work section (Section 3.5), precise string analysis using static analysis is hard, especially for applications that use complex constructs to manipulate strings or interact with external functions to compute strings that are used in queries.

3.1.2 Our Approach

To deduce at run-time the query structure intended by a programmer, our high-level idea is to dynamically construct the structure of the programmer-intended query whenever the execution reaches a program location that issues a SQL-query. Our approach is to compute the intended query by running the application on *candidate* inputs, that are self-evidently non-attacking. For the above example, the candidate input for variable `name` set to "`John`" and the variable `modify` set to `false`, elicits the intended query along the branch that enters the first `if-then` block (`Path 1`) in Figure 3.2. In order for a candidate input to be useful, it must satisfy the following two conditions:

1. *Inputs must be benign*. The candidate input must be evidently non-attacking, as envisioned by the programmer while coding the application. The queries constructed from these inputs, therefore, will not be attack queries.

2. *Inputs must dictate the same path that was dicated by the actual inputs*. An actual input to the program will dictate a control path to a point where a query is issued. To deduce the programmer-intended query structure for this particular path (i.e., the *control path*), the candidate inputs must also exercise the same control path in the program.

Given such candidate inputs, we can detect attacks by comparing the query structures of the programmer-intended query (computed using the candidate input) and the possible attack query.

The above discussion suggests the need for an *oracle* that, given a control path in a program, returns a set of benign candidate inputs that exercise the same control path. This oracle, if constructed, may actually offer us a clean solution to the problem of deducing the query structure intended by the programmer. Unfortunately, such an oracle is hard to construct, and is, in the general case, impossible (i.e., the problem of finding such candidate inputs is *undecidable*). Indeed, if construction of such an oracle was indeed achievable, then one could construct a decision procedure for the halting problem by asking the oracle if at all there exists a set of inputs that dictate control to any halt instruction in the program.

The crux of our approach is to *avoid* the above problem of finding candidate inputs that exercise a control path, and instead derive the intended query structure directly from the same control path. We suggest that we can simply *execute* the statements along the control path on *any* benign candidate input, *ignoring* the conditionals that lie on the path. In the above example, Path 1 is taken for the attack input `John' OR 1=1 --'`. We can execute the statements along that path, in this case the lone `SELECT` statement, using the benign input "`John`" and dynamically discover the programmer intended query structure for the same path.

The idea of executing the statements on a control path, but not the conditionals along it, is, to the best of our knowledge, a new idea. It is in fact a very intuitive and theoretically sound approach, as shown by our formal description in the next section. Intuitively, when a program is run on an actual input, it exercises a control path, and the query constructed on that path can be viewed as a *symbolic* expression parameterized on the input variables. A natural approach to compute the intended query is then to substitute benign candidate inputs in the *symbolic*

expression for the query. This substitution is (semantically) precisely the same as evaluating the non-conditional statements on the control path on the candidate input, as shown in the next section.

3.2 Formal Analysis Using Symbolic Queries

In this section, we formalize SQL injection attacks and, through a series of gradual refinements and approximations, we derive the detection scheme used by CANDID. In order to concentrate on the main ideas in this analytic section, we will work with a simple programming language.

P	::=	defn ; stmt; `sql.execute`(s_0)	(program)
defn	::=	`int` n | `str` s | `input int` In | `input str` Is | defn ; defn	(variable declaration)
stmt	::=	stmt ; stmt | $n := ae$ | $s := se$ | `skip` | `while` (be) {stmt} | `if` (be) `then` {stmt} `else` {stmt}	(statement)
ae	::=	$c \mid n \mid f_i(t_1, \ldots, t_k)$	(arithmetic expressions)
se	::=	$cstr \mid s \mid g_i(t_1, \ldots, t_k)$	(string expressions)
be	::=	`true` | `false` | $h_i(t_1, \ldots, t_k)$	(boolean expressions)

where $n \in \mathcal{I}$, $s \in \mathcal{S}$, $c \in \mathbb{Z}$ is any integer constant, $cstr$ is any string constant, each t_i is either ae, be or se, depending on the parameters for f_i, g_i, h_i, respectively.

Figure 3.3: A Simple While Language

We first define a simple while-programming language (see Fig. 3.3) that has only two variable domains: integers and strings. We fix a set of integer variables $\mathcal{I}$ and a set of string variables $\mathcal{S}$, and use n, n_i, etc., to denote integer variables and s, s_j, etc., to denote string variables. A subset of these are declared to be *input* variables using the keyword `input`, and is used to model user-inputs to a web application.

Let us also fix a set of functions f_i each of which take a tuple of values, each parameter being a string/integer/boolean, and return an integer. Likewise, let's fix a set of functions g_i (respectively h_i) that take a tuple of string/integer/boolean values and return a string (respectively boolean). A special string s_0 is the query string, and a special command `sql.execute`(s_0) formulates an SQL-query to a database; we will assume that the query occurs exactly once and is the last instruc-

tion in the program. The syntax of programs is given in Fig. 3.3. The semantics is the natural one (and we assume each non-input integer variable is initialized to 0 and each non-input string variable is initialized to the null string ϵ).

Note that the functions f_i, g_i, and h_i are native functions offered by the language, and include arithmetic functions such as addition and multiplication, string functions such as concatenation and sub-string, and string-to-integer functions such as finding the length of a string.

For example, if `concat` is a function that takes two strings and concatenates them, then a string expression of the form:

```
concat(`` SELECT * FROM employdb WHERE name='', s)
```

denotes the concatenation of the constant string with the variable string s. For readability, however, we will represent concatenation using '$+$' (eg. "`SELECT * FROM employdb WHERE name='`" $+$ s).

The formal development of our framework will be independent of the functions the language supports. For the technical exposition in this section, we will assume that all functions are *complete* over their respective domains.

Program P
`input int` n; `input str` s; `str` s_0;
`if` $(n = 0)$ `then` ;; Path 1
{s_0:="SELECT * FROM employdb WHERE name=' "+s+"' "}
`else` ;; Path 2
{s_0:="SELECT * FROM employdb WHERE name="+s+ " ' AND status='cur' "};
`sql.execute(`s_0`)`

Figure 3.4: An Example Program

Figure 3.4 illustrates a program that will serve as the running example throughout this section. The program takes an integer n and a string s as input, and depending on whether n (which could be a check-box in a form) is 0 or not, forms a dynamically generated query s_0. Note that the query structures generated in the two branches of the program are different. The input determines the control path taken by the program, which in turn determines the structure of the query generated by the program.

3.2.1 SQL Injection Defined

Let us assume a standard syntax for SQL queries, and define two queries q and q' to be equivalent (denoted $q \approx q'$) if the parse structures of the two queries is the same. In other words, two queries are equivalent if the parse trees of q and q' are isomorphic.

Let P be a program with inputs $I = \langle In_1, \ldots In_p, Is_1, \ldots Is_q \rangle$. An input valuation is a function v that maps each In_j to some integer and maps each Is_j to some string. Let IV denote the set of all input valuations. For any input valuation v, the program P takes a unique control path Run_v (which can be finite, or infinite if P doesn't halt). We will consider only halting runs, and hence Run_v will end with the instruction `sql.execute(`s_0`)`. Note that the path Run_v, and hence the structure of the query s_0, could depend on the input valuation v (e.g., due to conditionals on input variables as in Fig. 3.4).

Intuitively, the input valuations are partitioned into two parts: the set of *valid* inputs V which are benign, and the complement $\overline{V}$ of *invalid* inputs, which include all SQL injection attacks. A definition of SQL injection essentially defines this partition.

The primary principles on which our definition of SQL injection is based on are the following:

- (**P1**) the structure of the query on any valid valuation v is determined solely by the control path the program takes on input v.
- (**P2**) an input valuation is invalid iff it generates a query structure different from the structure determined by its control path.

The principle (P1) holds for most practical programs that we have come across as well as any natural program we tried to write. An application, such as the one in Figure 3.4, typically will generate different query structures depending on the input (the input value of the variable n, in this case). However, these query structures are generated differentially using conditional clauses that check certain values in the input (typically check-boxes in Web application input), as in in Figure 3.1. As shown in our comparison studies in Section 3.5, (P1) is actually a more general principle than the underlying ideas suggested in earlier works [14, 12].

Let v be an input and $\pi = Run_v$ be the path the program exercises on it. By (P1), we know that there is a unique query structure associated with π. (P2) says that every invalid input disagrees with this associated query structure. As mentioned in the earlier section, `PREPARE` statements are based on (P2). Moreover, (P2) is a well-agreed principle in other works on detecting SQL injection [17, 12]. Given the above principles, we can define SQL injection.

Let us first assume that we have a *valid representation* function $VR : IV \rightarrow V$, which for any input valuation v, associates a *valid* valuation v' that exercises the *same* control path as v does, i.e., if $VR(v) = v'$, then $Run_v = Run_{v'}$. Intuitively, the range of VR is a set of *candidate inputs* that are benign and exercise all feasible control paths in the program. This function may not even exist and is hard to statically or dynamically determine on real programs; we will circumvent the construction of this function in the final scheme.

Now we can easily define when an input v is invalid: v is invalid iff the structure computed by the program on v is different from the one computed on $VR(v)$.

Definition 1. *Let P be a program and $VR : IV \rightarrow V$ be the associated valid-representation function. An input valuation v for P is an* SQL-injection attack *if the structure of the query q that P computes on v is different from that of the query q' that P computes on $VR(v)$ (i.e., $q \not\approx q'$).*

Turning back to our example in Fig. 3.4, the input $v : \langle n \leftarrow 0; s \leftarrow$ "$Jim' \; OR \; 1 = 1 \; -\; -''\rangle$ is an SQL-injection attack since it generates a query whose structure is:

```
SELECT ? FROM ? WHERE ?=?  OR ?=?
```

Its corresponding candidate input $VR(v) = v_1 : \langle n \leftarrow 0; s \leftarrow$"John"$\rangle$ exercises the same control path and generates a different query structure:

```
SELECT ? FROM ? WHERE ?=?
```

Alternate Definition Using Symbolic Expressions

Let us now reformulate the above definition of SQL injection in terms that explicitly capture the *symbolic expression* for the query at the end of the run Run_v. Intuitively, on an input valuation v, the program exercises a path that consists of a set of assignments to variables. The symbolic expression for a variable summarizes the effect of all these assignments using a single expression and is solely over the input variables $\langle In_1, \ldots In_p, Is_1, \ldots Is_q\rangle$.

For example, consider the input v : $\langle n \leftarrow 0$ and $s \leftarrow$ "Jim' OR 1=1- -" $\rangle$ for the program in Fig 3.4. This input exercises Path 1, and the `SELECT` statement is the only statement along this path. The symbolic expression for the query string s_0 on this input at the point of query is $Sym_\pi(s_0)$:

```
``SELECT * FROM employdb WHERE name=' ''+s+`` ' ''
```

Definition 2 (Symbolic expressions)**.** *Let U be a set of integer and string variables, and let π be a sequence of assignments involving only variables in U. Then the symbolic expression after π for any integer variable $n \in V$ is an arithmetic expression, denoted $Sym_\pi(n)$, and the symbolic expression for a string variable $s \in V$ is a string expression, denoted $Sym_\pi(s)$. These expressions are defined inductively over the length of π using the rules given in Figure 3.5.*

- If $\pi = \epsilon$ (i.e., for the empty sequence),
  $Sym_\epsilon(n) = n$ if $n \in I$
  $= 0$ otherwise
  $Sym_\epsilon(s) = s$ if $s \in I$
  $= \epsilon$ otherwise

- If $\pi = \pi'.(n' := ae(t_1, \ldots, t_k))$, then
  $Sym_\pi(n) = Sym_{\pi'}(n)$, for every $n \in U, n \neq n'$
  $Sym_\pi(n') = ae(Sym_{\pi'}(t_1), \ldots, Sym_{\pi'}(t_k))$
  $Sym_\pi(s) = Sym_{\pi'}(s)$, for every $s \in U$.

- If $\pi = \pi'.(s' := se(t_1, \ldots, t_k))$, then
  $Sym_\pi(n) = Sym_{\pi'}(n)$, for every $n \in U$
  $Sym_\pi(s') = se(Sym_{\pi'}(t_1), \ldots, Sym_{\pi'}(t_k))$
  $Sym_\pi(s) = Sym_{\pi'}(s)$, for every $s \in U, s \neq s'$.

Figure 3.5: Inductive Rules for the Compuation of Symbolic Expressions

For an input valuation v, let π_v denote the set of *assignments* that occur along the control path Run_v that v induces, i.e., π_v is the set of statements of the form $n := ae$ or $s := se$ executed by P on input valuation v. Then the symbolic expression for the query s_0 on v is defined to be $Sym_{\pi_v}(s_0)$.

Observe that for any program P and input valuation v, the value of any variable x computed on v is $Sym_{\pi_v}(x)$. That is, the value of any variable can be obtained by substituting the values of the input variables in the symbolic expression for that variable.

Note that if v and v' induce the same run, (i.e., $Run_v = Run_{v'}$), then $\pi_v = \pi_{v'}$, and hence the symbolic expression for the query computed for v is *precisely* the same as that computed for v'.

We can hence reformulate SQL injection as in Definition 1 precisely as:

Definition 3. *Let P be a program and $VR : IV \rightarrow V$ be the associated valid-representation function. An input valuation v for P is an* SQL-injection attack *if the symbolic expression for the query, $exp = Sym_{\pi_v}(s_0)$ when evaluated on v has a different query structure than when evaluated on $VR(v)$ (i.e., $exp(v) \not\approx exp(VR(v))$).*

Consider the benign candidate input: $v_1 : \langle n \leftarrow 0$ and $s \leftarrow$"John"$\rangle$ corresponding to the input $v : \langle n \leftarrow 0$ and $s \leftarrow$"Jim' OR 1=1--" $\rangle$ for the program in Fig 3.4 as they exercise the same path (Path 1). The symbolic expression for s_0 on this valid input at the point of query is $Sym_{\pi}(s_0)$:

```
``SELECT * FROM employdb WHERE name=' ''+s+`` ' ''
```

Note that the *conditionals* that are checked along the control path are *ignored* in this symbolic expression. Substituting any input string for s tells us exactly the query computed by the program along this control path. Consequently, we infer that the input v is an SQL-injection attack since it follows the same path as the valid input above, but the structure of the query obtained by substituting $s \leftarrow$ "John" in the symbolic expression is *different* from that obtained by substituting $s \leftarrow$"Jim' OR 1=1--".

Observe that the solution presented by the above definition is hard to implement. Given an input valuation v, we can execute the program P on it, extract the path followed by it, and compute the symbolic expression along v. Now if we *knew* another valuation v' that exercised the same control path as v does, then we can evaluate the symbolic expression on v and v', and check whether the query structures are the same. However, it is very hard to find a valid input valuation that exercises the same path as v does.

Approximating the SQL injection problem

The problem of finding for every input valuation v, a corresponding valid valuation that exercises the same path as v does (i.e., finding the function VR) is a

hard problem. We now argue that a simple approximation of the above provides an effective solution that works remarkably well in practice.

We propose to simply drop the requirement that v' exercises the same control path as v. Instead, we define $VR(v)$ to be the valuation v_c that maps every integer variable to 1 and every string variable s to $a^{v(s)}$, where a^i denotes a string of a's of length i. That is, v_c maps s to a string of a's precisely as long as the string mapped by v.

We note that v_c is manifestly benign and non-attacking for *any* program. Hence substituting this valuation in the symbolic query must yield the *intended query structure* for the control-path executed on v. Consequently, if this intended query structure does not match the query the program computes on v, then we can raise an alarm and prevent the query from executing.

The fact that the candidate valuation v_c may not follow the same control path as v is not important as in any case we will not follow the control path dictated by v_c, but rather simply substitute v_c in the symbolic expression obtained on the control path exercised on v. Intuitively, we are *forcing* the program P to take the same path on v_c as it does on v to determine the intended query structure that the path generates. We will justify this claim using several practical examples below.

Consider our running example again (Figure 3.4). The program on input v : $\langle n \leftarrow 0, s \leftarrow$ "$Jim'\ OR\ 1 = 1\ -\ -$"$\rangle$ executes the `if`-block, and hence generates the symbolic query exp:

"`SELECT * FROM employdb WHERE name='`" + s + "`'`"

Substituting the input values in this expression yields

`Q1:SELECT * FROM employdb WHERE name='Jim' OR 1=1--'`

Consider the valuation v_c : $\langle n \leftarrow 1, s \leftarrow$ "$aaaaaaaaaa$" $\rangle$. The program on this path follows a *different* control path (going through the `else`-block), and generates a query whose structure is quite unlike the query structure obtained by pursuing the `if`-block. However, substituting v_c in the symbolic expression exp yields

`Q2:SELECT * FROM employdb WHERE name='aaaaaaaaaa'`

which is indeed the correct query structure on pursuing the `if`-block. Since the query structures of `Q1` and `Q2` differ, we detect that the query input is an SQL-injection attack. Note that an input assigning $\langle n \leftarrow 0, s \leftarrow$ "$Jane$"$\rangle$ *will match* the structure of the candidate query.

The above argument leads us to an approximate notion of SQL injection:

Definition 4. *Let P be a program, and v be an input valuation, and v_c the benign candidate input valuation corresponding to v. An input valuation v for P is a* SQL-injection attack *if the symbolic expression exp for the query string s_0 on the path exercised by v results in different query structures when evaluated on v and v_c (i.e., $exp(v) \not\approx exp(v_c)$).*

The above scheme is clearly implementable as we can build the symbolic expression for the query on the input to the program, and check the structure of the computed query with the structure of the query obtained by substituting candidate non-attacking values in the symbolic query. However, we choose a simpler way to implement this solution: we transform the original program into one that at every point computes values of variables both for the real input as well as the candidate input, and hence evaluates the symbolic query on the candidate input in tandem with the original program.

3.3 The CANDID Transformation

In this section, we discuss the transformation procedure for the dynamic candidate evaluation technique described in the earlier section. We accomplish this using a simple program transformation of the web application.

For every string variable v in the program, we add a variable v_c that denotes its candidate. When v is initialized from the user-input, v_c is initialized with a benign candidate value that is of the same length as v. If v is initialized by the program(e.g. by a constant string like an SQL query fragment), v_c is also initialized with the same value. For every program instruction on v, our transformed program performs the same operation on the candidate variable v_c. For example, if x and y are two variables in the program, the operation:

`"SELECT * FROM employdb WHERE name="` $+ x +$ y

results in the construction of a query, or a partial query string. The transformer performs a similar operation immediately succeeding this operation on the candidate variables:

`"SELECT * FROM employdb WHERE name="` $+ x_c +$ y_c

The operation on the candidate variables thus mirror the operations on their counterparts. The departure to this comes while handling conditionals, where the

```
1  void process-form(String uname, String uname_c, String
       pwd, String pwd_c, boolean modify, String phonenum
     , String phonenum_c) {
2   if (modify == false){ /* Path 1. only display */
3    query = "SELECT * FROM phonebook WHERE username = '"
        + uname + "' AND password = ' " + pwd + "'";
4    query_c = "SELECT * FROM phonebook WHERE username =
       '" + uname_c + "' AND password = ' " + pwd_c + "'
       ";
5   }
6   else{ /* modify telephone number */
7     if (phonenum == ""){ /* Path 2. delete entry */
8       query = "DELETE FROM phonebook WHERE username='"
          + uname + "' AND password = '" + pwd + " ' ";
9       query_c = "DELETE FROM phonebook WHERE username='
          " + uname_c + "' AND password = '" + pwd_c + "
           ' ";
10     }else{ /* Path 3. update entry */
11      query = "UPDATE phonebook SET phonenumber =" +
          phonenum + "WHERE username = ' "  + uname + "'
           AND password = '" + pwd + "'";
12      query_c = "UPDATE phonebook SET phonenumber =" +
          phonenum_c + "WHERE username = ' " + uname_c +
           "' AND password = '" + pwd_c + "'";
13    }
14  }
15  compare-parse-trees(query,query_c); /* throw
      exception if no match */
16  sql.execute(query);
17 }
```

Figure 3.6: Transformed Source for the Example in Figure 3.2

candidate computation needs to be forced along the path dictated by the real inputs. Therefore, our translator does not modify the condition expression on the `if-then-else` statement. At run-time, the conditional expression is then only evaluated on the original program variables, and therefore dictates candidate computation along the actual control path taken by the program. The transformation for the `while` statements are similar.

Function calls are transformed by adding additional arguments for candidate samples. Due to the type safety guarantees of our target language (Java), we only

maintain candidates for string variables. We also do not transform expressions that do not involve string variables. For those expressions that involve use of non-string variables in string expressions, we directly use the original variable's values for the candidate.

The transformation for the SQL query statement `sql.execute` calls a procedure `compare-parse-trees` that compares the real and candidate parse trees. This procedure throws an exception if the parse trees are not isomorphic. Otherwise, the original query is issued to the database.

Figure 3.6 gives the transformed code for the program illustrated in Figure 3.2. The actual transformation rules for the while language presented in the previous section is presented in Figure 3.7.

3.3.1 Resilience of CANDID

The transformation of programs to dynamically detect intentions of the programmer using candidate inputs as presented above is remarkably resilient in a variety of scenarios. We outline some interesting input manipulations Web applications perform, and illustrate how CANDID handles them. Several approaches in the recent literature for preventing SQL injection attacks fail in these simple scenarios (see Section 3.5).

Conditional queries. Conditional queries are differential queries constructed by programs depending on predicates on the input. For example, a program may form different query structures depending on a boolean input (such as in Figure 3.2), or perhaps even on particular values of strings. The candidate input may not match the real queries on these predicates, and hence may take a different path than the real input. However, in the CANDID approach, conditionals are always evaluated on the real inputs only, and hence the candidate query is formed using the same control path the real input exercises. An illustrative example: consider a program that issues an INSERT-query if the input string `mode` is "ADD" and issues a DELETE-query if `mode` is "MODIFY". For a real query with `mode`="ADD", the candidate query will assign `mode`="`aaa`" which, being an invalid mode, may actually cause the program to throw an exception. However, the test for `mode` is done on the real string and hence the candidate query will be an INSERT-query with appropriate values of the candidate input substituted for the real input in the query.

	Grammar Production	Definition of the function Γ()	
defn	→ `int` n	{ `int` n }	(1a)
	| `str` s	{ `str` s_c; `str` s }	(1b)
	| defn$_1$; defn$_2$	{ Γ(defn$_1$) ; Γ(defn$_2$) }	(1c)
	| `input-int` n	{ `input-int` n }	(1d)
	| `input-str` s	{ `input-str` s ; `str` s_c := `str-candidate-val(s)` }	(1e)
stmt	→ `skip`	{ `skip` }	(2a)
	| $s := se$	{ s_c := Γ(se) ; $s := se$ }	(2b)
	| $n := ae$	{ $n := ae$ }	(2c)
	| stmt$_1$; stmt$_2$	{ Γ(stmt$_1$); Γ(stmt$_2$) }	(2d)
	| `if` be stmt$_1$ `else` stmt$_2$	{ let t-stmt$_1$ = Γ(stmt$_1$) in let t-stmt$_2$ = Γ(stmt$_2$) in `if` be t-stmt$_1$ `else` t-stmt$_2$ }	(2e)
	| `while` be stmt$_1$	{ let t-stmt$_1$ = Γ(stmt$_1$) in `while` be t-stmt$_1$ }	(2f)
ae	→ c	{ c }	(3a)
	| n	{ n }	(3b)
	| f_i(t$_1$,. . .,t$_k$)	{ f_i(t$_1$,. . .,t$_k$) }	(3c)
se	→ $cstr$	{ $cstr$ }	(3d)
	| s	{ s_c }	(3e)
	| g_i(t$_1$,. . .,t$_k$)	{ g_i(Γ(t$_1$),. . .,Γ(t$_k$)) }	(3f)
be	→ `false`	{ `false` }	(3g)
	| `true`	{ `true` }	(3h)
	| h_i(t$_1$,. . .,t$_k$)	{ h_i(t$_1$,. . .,t$_k$) }	(3i)
`sql.execute(`se`)`	→ `sql.execute(`se`)`	{ let t-se = Γ(se) in compare-parse-trees(se,t-se); `sql.execute(`se`)` }	(4)

Figure 3.7: Transformation To Compute Candidate Queries

Input-splitting. Programs may not atomically embed inputs into queries. For example, a program may take an input string `name`, which contains two words, such as "Alan Turing", and may issue a SELECT query with the clauses `FIRSTNAME='Alan' and LASTNAME='Turing'`. In this case, the candidate input is a string of a's

of length 11, and though it does not have any white-space, the conditional on where to split the input is done on the real query, and the candidate query will have the clauses `FIRSTNAME='aaaa' and LASTNAME='aaaaaa'`, which elicits the intended query structure.

Preservation of lengths of strings. The method of forcing evaluation of candidate inputs along the control path taken by the real input may at first seem delicate and prone to errors. An issue is that the operations performed on the candidate path may raise exceptions. The most common way this can happen is through string indexing: the program may try to index into the i'th character of a string, and this may cause an exception if the corresponding string on the candidate evaluation is shorter than i. This is the reason why we choose the candidate inputs to be precisely the same length as the real inputs. Moreover, for all relevant string operations we can show that the lengths of the real and candidate strings are preserved to be equal. More precisely, consider a function g that takes strings and integers as input and computes a string. The function g is said to be *length preserving*, if the length of the string returned by g as well as whether g throws an exception depends only on the *lengths* of the parameter strings and the values of the integer variables. All string functions in the Java String class (such as concatenation and substring function) are in fact length-preserving. We can show that the strings for candidate variables are precisely as long as their real variable counterparts across any sequence of commands and calls to length-preserving functions. Therefore, they will not throw any exception on the candidate evaluation. In all the experiments we have conducted, the candidate path never raises an exception.

External functions and stored queries. CANDID also handles scenarios where external functions and stored queries are employed in a program. When an external function ext (for which we do not have the source) is called, as long as the function is free of side-effects, CANDID safely calls ext twice, once on the real variables and once on the candidate variables. Methods such as tainting are infeasible in this scenario as tracking taints cannot be maintained on the external method; however, CANDID can still keep track of the real and intended structures using this mechanism.

Stored queries are snippets of queries stored in the database or in a file, and programs use these snippets to form queries dynamically. Stored queries are commonly used to maintain changes to the database structure that are dynamically

changed over time to reflect changes in configurations. Changes to stored queries pose problems for static methods as the code requires a fresh analysis, but poses no problems to CANDID as it evaluates attacks dynamically on each run.

3.4 Implementation and Evaluation

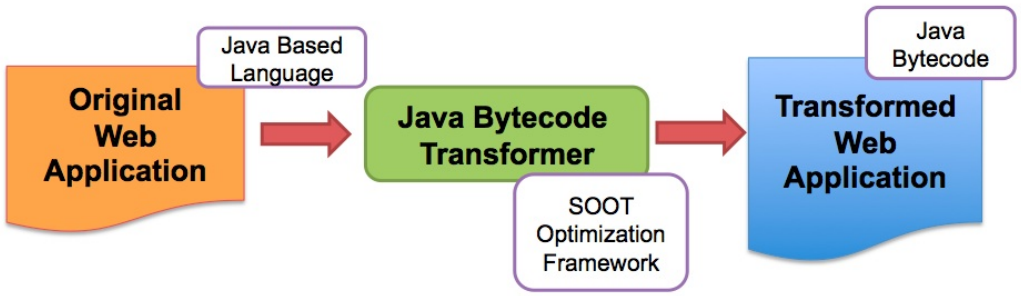

Figure 3.8: Offline View of CANDID

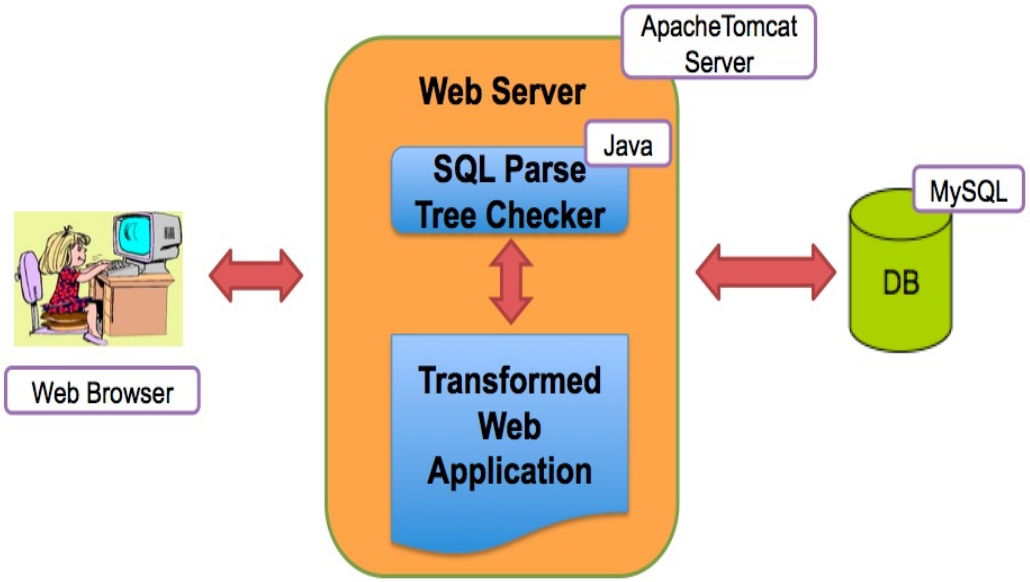

Figure 3.9: Run-time View of CANDID

Our tool, CANDID, is implemented to defend applications written in Java, and works for any web application implemented through Java Server Pages or as Java

servlets. Figures 3.8 and 3.9 give an overview of our implementation. CANDID consists of two components: an offline Java program transformer that is used to instrument the application, and an (online) SQL parse tree checker. The program transformer is implemented using the SOOT [55] Java transformation tool. The SQL parse-tree checker is implemented using the JavaCC parser-generator.

3.4.1 Transformation

The automated transformation was implemented for Java byte-code using an extension to the SOOT optimization framework [55]. SOOT provides a three-address intermediate byte-code representation, Jimple, suitable for code analysis and optimization. Class files of the uninstrumented applications were processed using the SOOT framework with CANDID to generate instrumented class files for deployment.

The transformer handles all fifteen types of Jimple statements e.g., `InvokeStmt, AssignStmt`, etc. If a statement is found to be acting on, producing or leading to `String` type objects, the transformer adds appropriate statements to perform candidate evaluation; for example, identity statements are used to pass parameters to methods. For user defined methods, corresponding to each String parameter, a candidate parameter is added to the method signature and an identity statement is inserted in the method body for parameter passing.

As mentioned earlier, we compare the parse trees of the real and candidate queries for attack detection. It is worthwhile to mention here that even the slightest mismatch of the parse trees is detected as an attack.

3.4.2 Application Examples

We evaluated our technique using a suite of applications that was obtained from an independent research group [14]. This test suite contained seven applications, five of which are commercial applications: Employee Directory, Bookstore, Events, Classifieds and Portal. These are available at `http://www.gotocode.com`. The two other applications, Checkers and Officetalk, were developed by the same research group. These applications were medium to large in size (4.5KLOC - 17KLOC).

Table 3.1 summarizes the statistics for each application. The number of servlets

Application	LOC	Servlets	SCL
Employee Dir	5,658	7 (10)	23
Bookstore	16,959	3 (28)	71
Events	7,242	7 (13)	31
Classifieds	10,949	6 (14)	34
Portal	16453	3 (28)	67
Checkers	5421	18 (61)	5
Officetalk	4543	7 (64)	40

Table 3.1: Applications from the test suite

in the second column gives the number exercised in our experiment, with the total number of servlets in brackets. Our goal was to perform large-scale automated tests (as described below), and some servlets could only be accessed through a complex series of interactions with the application that involved a human user, and therefore were not exercised in our tests. The column SCL reports the number of SQL Command Locations, which issue either a `sql.executeQuery` (mainly `SELECT` statements) and `sql.executeUpdate` (consisting of `INSERT`, `UPDATE` or `DELETE` statements) to the database. Immediately preceding this command location, the CANDID instrumentation calls the parse tree comparison checker.

3.4.3 Attack Suite

The attack test suite was also obtained from the authors of [14]. It consists, for each application, both attack and non-attack inputs that test several kinds of SQL injection vulnerabilities. Overall, the attack suite contains 30 different attack string patterns (such as tautology-based attacks, UNION SELECT based attacks [13]), that were constructed based on real attacks obtained from sources US/CERT and CERT/CC advisories. Based on these attack strings, the attack test suite makes use of each servlet's injectible web inputs.

The test suite also contained non-attack (benign) inputs that tested the resilience of the application on legitimate inputs that "looked like" attack inputs. These inputs contain data that may possibly break the application in the face of SQL input validation techniques.

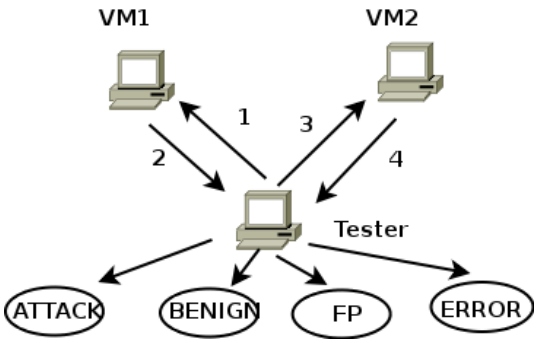

Figure 3.10: Testbed Setup

3.4.4 Experiment Setup

Our objective was to deploy two versions of each application: (1) an original uninstrumented version and (2) a CANDID protected version. Also, to simulate a live-test scenario, we wanted to deploy attacks simultaneously on each of these two versions and observe the results. We wanted the original and instrumented versions to be isolated from each other, so that they do not affect the correctness of tests. For this reason we decided to run them on two separate machines.

In order that the state of the two machines be the same at the beginning of the experiments, we adopted the following strategy: On a host RedHat Linux system, we created a virtual machine running on VMware also running RedHat EL 4.0 guest operating system. We then installed all the necessary software in this virtual machine: the Apache webserver and Tomcat JSP server, MySQL database server, and the source and bytecode of all Java web applications (original and instrumented versions) in our test suite. Through an automated script, we also populated the database with tables required by these applications. After configuring the applications to deployment state, we *cloned* this virtual machine by copying all the virtual disk files to another host machine with similar configurations. This resulted in two machines that were identical except for the fact that they ran the original and instrumented versions of the web applications.

Figure 3.10 shows the test scenario. The original application was deployed on virtual machine VM1 and the instrumented application was deployed on virtual machine VM2. A third machine ("Tester") was used to launch the attacks over HTTP on the original and instrumented web applications, and also was used to immediately analyze the result. For this purpose, a suite of Perl scripts utilizing the `wget` command were developed and used. The master script that ran the at-

tack scripts ran in the following sequence, as shown in the figure (1) it launched the attack first on the original application and (2) recorded the responses. It then (3) launched the attack on the instrumented application and (4) recorded the responses. After step (4), another post-processing script compared the output from the two VMs and classified the result into one of the following cases (a) the attack was successfully detected by the instrumented application (b) the instrumented application reported a benign string as an attack (c) the instrumented application reported a benign string as benign (d) errors were reported by the original or instrumented application.

3.4.5 Attack Evaluation

We ran the instrumented application with the attack suite, and the results are summarized in Table 3.2. The second column lists the number of input attempts, and the third lists the number of successful attacks on the original application. The number of attacks detected by the instrumented application is shown in the same column. The fourth column shows the number of non-attack benign inputs and any false positives for the instrumented application. CANDID instrumented applications were able to defend all the attacks in the suite, and there were zero false positives.

The test suite we received had a large number of attack strings that resulted in invalid SQL queries and are reported in column 5. We used a standard SQL parser based on SQL ANSI 92 standard, augmented with MYSQL specific language extensions. To ensure the correctness of our parser implementation, we verified that these queries were in fact malformed using an online SQL Query formatter [56].

3.4.6 Performance Evaluation

For testing the performance impact we used the web application benchmarking tool JMeter [57] from Apache Foundation, an industry standard tool for measuring performance impact on Java web applications.

We computed the overhead imposed by the approach on one servlet that was chosen from each application, and prepared a detailed test suite for each application. As typical for web applications, the performance was measured in terms of differences in response times as seen by the client. The server was on a Red

Application	Input attemp.	Succ. Attacks	FPs/ Non-attacks (Benign)	Parse Errors
EmployeeDir	7,038	1529/1529	0/2442	3067
Bookstore	6,492	1438/1438	0/2930	2124
Events	7,109	1414/1414	0/3320	2375
Classifieds	6,679	1475/1475	0/2076	3128
Portal	7,483	2995/2995	0/3415	1073
Checkers	8,254	262/262	0/7435	557
Officetalk	6,599	390/390	0/2149	4060

Table 3.2: Attack Evaluation results

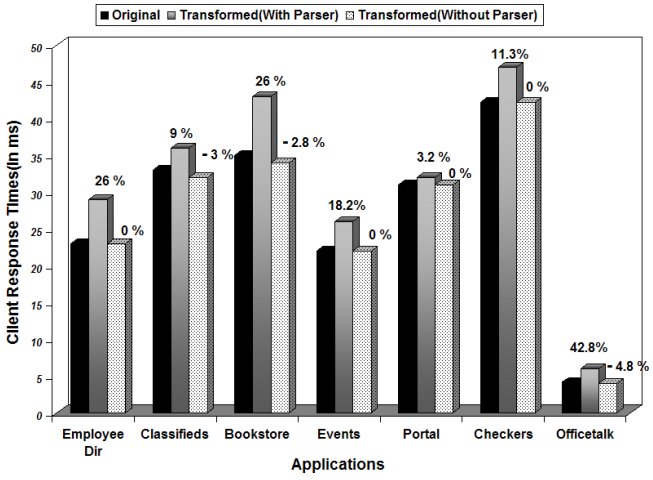

Figure 3.11: Performance Overhead

Hat Enterprise Linux machine with a 2GHz Pentium processor and 2GB of RAM, that ran in the same Ethernet network as the client. Note that this scenario does not have any network latencies that are typical for many web applications, and is therefore an indicator of the worst case overheads in terms of response times.

For each test, we took 1000 sample runs and measured the average numbers for each run, with caching disabled on the JSP / Web/ DB servers. The results are shown in Figure 3.11. The figure depicts the time taken by the original application, the transformed code, and also the transformed code without the parser component.

Figure 3.11 indicates that instrumented applications without SQL parser calls

had negligible overhead over the original applications (also optimized for performance using SOOT), when compared to uninstrumented applications. Figure 3.11 also indicates that instrumented applications with SQL parser code had varying overheads and ranged from 3.2% (for Portal application) to 40.0% (for OfficeTalk application). These varying overheads are mainly attributed to varying numbers of SQL parser calls in each page e.g., Bookstore application invoked SQL parser code 7 times for the selected page, whereas Portal application only invoked it once. OfficeTalk application's high overhead is attributed to the fact that client response time for uninstrumented application is very small (5ms) when compared to other applications (23ms - 39ms). This application's actual execution time is dwarfed by factors like class load time and resulted in high overhead for instrumented application. In other applications actual execution time is considerable, and thus the overheads are significantly less.

The above results clearly show that CANDID's overheads are quite acceptable. We briefly discuss some of the possible optimizations here that could further improve the performance. The SQL parser contributes to most of the overhead and can be greatly improved. Notably, the two class files of SQL parser we used are large– 54KB and 21KB, and are frequently loaded. A hand coded SQL parser can be significantly smaller in size. Also, by performing flows-to/reachability analysis, we can avoid transformations of string operations that do not contribute to the query.

3.5 Related Work

There has been intense research in detection and prevention mechanisms against SQL injection attacks recently. We can classify these approaches broadly under three headings: (a) coding practices that incorporate defensive mechanisms that can be used by programmers, (b) vulnerability detection using static analysis techniques that warn the programmer of possible attacks, and (c) *defensive* techniques that detect vulnerabilities and simultaneously prevent them.

Defensive coding practices include extensive input validation and the usage of `PREPARE` statements in SQL. Input validation is an arduous task because the programmer must decide the set of valid inputs, escape special characters if they are allowed (for example, a name-field may need to allow quotes, because of names like `O'Niel`), must search for alternate encodings of characters that encode SQL

commands, look for presence of back-end commands, etc. `PREPARE` statements semantically separate the role of keywords and data literals. Using `PREPARE` statements is very effective against attacks and is likely to become the standard prevention mechanism for freshly written code; augmenting legacy programs to prepare statements is hard to automate and not viable. Two similar approaches, SQL DOM [58] and Safe Query Objects [59], provide executable mechanisms that enable the user to construct queries that isolate user input.

3.5.1 Vulnerability Detection Using Static Analysis

There are several approaches that rely solely on static analysis techniques [8, 7] to detect programs vulnerable to SQLCIA. These techniques are limited to identifying sources (points of input) and sinks (query issuing locations), and checking whether every flow from a source to the sink is subject to input validation ([8] is flow-insensitive while [7] is flow-sensitive). Typical precision issues with static analysis, especially when dealing with dynamically constructed strings, mean that they may identify several such illegal flows in a web application, even if these paths are infeasible. In addition, the user must *manually* evaluate and declare the sanitizing blocks of code for each application, and hence the approach is not fully automatable. More importantly, the tools do not in any way help the user determine whether the sanitization routines prevent all SQL injection attacks. Given that there are the various flawed sanitization techniques for preventing SQL injection attacks (several myths abound on Internet developer forums), we believe there are numerous programs that use purported sanitization routines that are not correct, and declaring them as valid sanitizers will result in vulnerable programs that pass these static checks.

3.5.2 Defensive Techniques that Prevent SQLCIA

Defensive techniques that prevent SQL injection attacks are significantly different from vulnerability analysis as they achieve the more complex (and more desirable) job of transforming programs so that they are guarded against SQL injection attacks. These techniques do not demand the programmer to perform input validation to stave off injection attacks, and hence offer effective solutions for securing applications, including legacy code. We discuss three approaches in detail be-

low; for a more detailed account of various other techniques and tools, including paradigms such as instruction set randomization [60], proxy filtering of input, and testing, we refer the reader to a survey of SQL injection and prevention techniques [13].

Learning programmer intentions statically. One approach in the literature has been to learn the set of all intended query structures a program can generate and check at run-time whether the queries belong to this set. The learning phase can be done statically (as in the AMNESIA tool [14]) or dynamically on test inputs in a preliminary learning phase [15]. The latter has immediate drawbacks: incomplete learning of inputs result in inaccuracies that can stop execution of the program on benign inputs.

A critique of AMNESIA: Consider a program that takes in two input strings `nam1` and `nam2`, and issues a select query that retrieves all data where the `name`-field matches either `nam1` or `nam2`. If `nam2` is empty, then the select query issues a search only for `nam1`. Further, assume the program ensures that neither `nam1` nor `nam2` are the string "`admin`" (preventing users from looking at the administrators entries). There are two intended query structures in this program:

```
"SELECT * FROM employdb WHERE name='" + nam1 + "'"

"SELECT * FROM employdb WHERE name='" + nam1 + "'" +
                "OR name='" + nam2 + "'"
```

with the requirement that neither `nam1` nor `nam2` is "admin".

We tested the Java String Analyser (the string analyzer used in AMNESIA [14] to learn query structures statically from Java programs) on the above example. First, JSA detected the above two structures, but could not detect the requirement that `nam1` and `nam2` cannot be "admin". Consider now an attack of the program where `nam1` = "`John' OR name='admin`" and `nam2` is empty. The program will generate the query:

```
SELECT * FROM employdb WHERE name='John'
                OR name='admin'
```

and hence retrieve the administrator's data from the database. Note that though the above is a true SQL injection attack, a tool such as AMNESIA would allow this as its structure is a possible query structure of the program on benign inputs.

The problem here is of course flow-sensitivity: the query structure computed by the program must be compared with the query structure the programmer intended along *that particular path* in the program. Web application programs use conditional branching heavily to dynamically construct SQL queries and hence require a flow sensitive analysis. The CANDID approach learns intentions dynamically and hence achieves more accuracy and is flow-sensitive.

Dynamic Tainting approaches. Dynamic approaches based on tainting input strings, tracking the taints along a run of the program, and checking if any keywords in a query are tainted before executing the query, are a powerful formalism for defending against SQL injection attacks.

Four recent taint platforms [10, 9, 11, 16] offer compelling evidence that the method is quite versatile across most real-world programs, both in preventing genuine attacks and in maintaining low false positives. The taint-based approach fares well on all experiments we have studied and several common scenarios we outlined in Section 3.3.1.

Our formalism is complimentary to the tainting approach. There are situations where the candidate approach results in better accuracy compared to the taint approach. Typical taint strategies require the source code of the entire application to track taint information. When application programs call procedures from external libraries or calls to other interpreters, the taint based approach requires these external libraries or interpreters to also keep track of tainting or make the assumption the return values from these calls are entirely tainted. The second choice may negatively impact tainting accuracy. In our approach, we can call the functions twice, one for the real input and one for the candidate input, which works provided the external function does not have side-effects.

Dynamic Bracketing approaches. Buehrer et al. [17] provide an interesting approach where the application program is *manually* transformed at program points where input is read, and the programmer explicitly brackets these user inputs (using random strings) and checks right before issuing a query whether any SQL keyword is spanned by a bracketed input. While this is indeed a very effective mechanism, it relies on the programmer to correctly handle the strings at various stages; for example if the input is checked by a conditional, the brackets must be stripped away before evaluating the conditional.

In [12], the authors propose both a formalization and an automatic transforma-

tion based on the above solution. The formalism is the only other formal definition of SQL injection in the literature, and formalizes changes of query structure using randomized bracketing of input. The automatic transformation adds random meta-characters that bracket the input, and adds functions that detect whether any bracketed expression spans an SQL-keyword. However, the formalism and solution set forth in [12] has several drawbacks:

• The solution of meta-bracketing may not preserve the semantics of the original program even on benign inputs. For example, a program that checks whether the input is well-formed (like whether a credit card number has 16 digits) may raise an error on correct input because of the meta-characters added on either side of the input string. There are several other scenarios outlined in Section 3.3.1 where the scheme fails: conditional querying (where say a string input determines the query structure, but would fail with meta-brackets), input splitting (since the input word would span across keywords), etc. Adding meta-characters only *after* such checks are done in the program is feasible in manual transformation [17] (though it would involve tedious effort), but is very hard to automate and sometimes impossible (for example if properties of the input are used later in the program, say when the input gets output in a tabular form where the width of tables depends on the length of the inputs).

• The above problems are in fact deep-rooted in the formalism developed in [12], which considers an overly simple notion of an application program that essentially takes in the input, applies a single filter function on it, and concatenates them to form a query. Program constructs such as conditionals and loops are ignored and is the source of the above problem (formally, a function applied on a bracketed input can behave very differently than when applied on the real input). Our formalism is much more robust in this regard and the definition of SQL injection in Definition 1 and Definition 3 are elegant and accurate definitions that work on realistic programs.

In summary, we believe that the dynamic taint-based approach and the CANDID approach presented in this chapter are the only techniques that promise a real scalable automatic solution to dynamically detect and prevent SQL injection attacks.

Acknowledgements This part of the thesis is joint work with Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. This research is supported in part by

NSF grants (CNS-0716584), (CNS-0551660), (IIS-0331707), (CNS-0325951), and CNS-0524695. We thank William Halfond and Alessandro Orso for providing us their test suite of applications and attack strings.

Chapter 4

Detection of cross-context scripting vulnerabilities in Browser Extensions

Driving the Internet revolution is the modern web browser, which has evolved from a relatively simple client application designed to display static data into a complex networked operating system tasked with managing many facets of a user's on-line experience. To help meet the varied needs of a broad user population, *browser extensions* expand the functionality of browsers by interposing on and interacting with browser-level events and data. Some extensions are simple and make only small changes to the appearance of web pages or the browser itself. Other extensions provide more sophisticated functionality, such as NoScript that provides fine-grained control over page JavaScript execution [61], or Greasemonkey that provides a full-blown programming environment for scripting browser behavior [62]. These are just a few of the thousands of extensions currently available for Firefox, the second most popular browser today.

Extensions written with benign intent can have subtle security-related bugs, called cross-context scripting vulnerabilities, that expose users to devastating attacks from the web, often just by viewing a web page. Firefox extensions run with full browser privileges, so attackers can exploit extension weaknesses to take over the browser, steal cookies or protected passwords, compromise confidential information, or even hijack the host system, without revealing their actions to the user. Unfortunately, dozens of extension vulnerabilities have been discovered in the last few years, and capable attacks against buggy extensions have already been demonstrated [63].

In this chapter, we propose Vex, a system for finding vulnerabilities in browser extensions using static information-flow analysis. Our key insight is that *extension vulnerabilities often translate into explicit information flows from injectable sources to executable sinks.* For extensions written with benign intent, most attacks involve the attacker injecting JavaScript into a data item that is subsequently executed by the extension under full browser privileges. We identify key flows of this nature that can lead to security vulnerabilities, and we check extensions for

the presence of such flows using a high-precision static analysis that is both path-sensitive and context-sensitive, to minimize the number of false positive suspect flows. VEX has special features to handle the quirks of JavaScript (e.g., VEX does a constant string analysis for expressions that flow into the `eval` statement that execute dynamically generated code).

Determining whether extensions are malicious or harbor security vulnerabilities is a hard problem. Extensions are typically complex artifacts that interact with the browser in subtle and hard to understand ways. For example, the ADBLOCK PLUS extension performs the seemingly simple task of filtering out ads based on a list of ad servers. However, the ADBLOCK PLUS implementation consists of over 11K lines of JavaScript code. Similarly, the NOSCRIPT extension provides fine-grained control over which domains are allowed to execute JavaScript and basic cross-site scripting protection. The NOSCRIPT extension implementation consists of over 19K lines of JavaScript code. Also, ADBLOCK PLUS had 41 releases in 1/1/06–6/10/11, and NOSCRIPT had 48 releases just in 1/1/11–6/10/11. While Mozilla uses volunteers to vet each new extension and revision before posting it on their official list of approved Firefox extensions, examining an extension to find a vulnerability requires a detailed understanding of the code to reason about anything beyond the most basic type of information flow. Thus tools to help vet browser extensions can be very useful for improving the security of extensions.

We show that VEX identifies 5 previously known vulnerabilities, and identifies other flows that led to the discovery of 7 previously unknown vulnerabilities, including vulnerabilities in the extensions WIKIPEDIA TOOLBAR, MOUSE GESTURES, and KAIZOU.

4.1 Threat Model, Assumptions, and Usage Model

In this article, we focus on finding security vulnerabilities in buggy browser extensions. We do not try to identify malicious extensions, bugs in the browser itself, or bugs in other browser extensibility mechanisms, such as plug-ins. We assume that the developer is neither malicious nor trying to obfuscate extension functionality, but we assume the developer could write incorrect code that contains vulnerabilities.

We use two attack models. First, we consider attacks that originate from web sites, and we assume the attacker can send arbitrary HTML and JavaScript to

the user's browser, modeling the usage model that assumes the user can navigate to any page on the internet. We focus on attacks where this untrusted data can lead to code injection or privilege escalation through buggy extensions. In the second attack model, we assume the same model as above, but we consider certain web sites as trusted. For example, if an extension gleans information from the Facebook website, we assume that the Facebook data will *not* include arbitrary HTML and JavaScript, but only well formatted and trusted data.

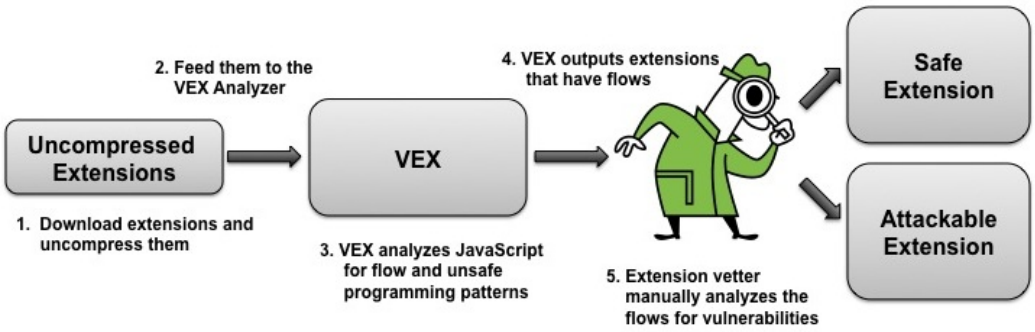

Figure 4.1: The overall analysis process of VEX.

According to the Mozilla developer site, Mozilla has a team of volunteers who help vet extensions manually. They run new and updated extensions isolated in a virtual machine to test the user experience. The editors also use a validation tool, which uses *grep* to look for key indicators of bugs. Many of the patterns they search for involve interactions between extensions and web pages, and they use their understanding of these patterns to help guide their inspection of the code. Our goal is to help automate this process, so that analysts can quickly hone in on particular snippets of code that are likely to contain security vulnerabilities. Figure 4.1 shows our overall work flow for using VEX: when extensions are subject to analysis by VEX, it reports precise code paths from untrusted sources to executable sinks in the extensions' code, which an expert must manually examine to check whether they can be used to mount an attack.

4.2 VEX Information Flow Patterns

Firefox has two privilege levels: *page* for the web page displayed in the browser's content pane, and *chrome* for elements belonging to Firefox and its extensions.

Page privileges are more restrictive than chrome privileges. For example, a page loaded from `illinois.edu` can only access content from `illinois.edu`. Firefox code and extensions run with full chrome privileges, which enables them to access all browser states and events, OS resources like the file system and network, and all web pages. Extensions also can include their own user interface components via a *chrome document*, which can run with full chrome privileges.

Firefox has APIs for extension code to communicate across protection domains and these interactions are one cause of extension security vulnerabilities. As the Mozilla developer site explains, "One of the most common security issues with extensions is execution of remote code in privileged context. A typical example is an RSS reader extension that would take the content of the RSS feed (HTML code), format it nicely and insert into the extension window. The issue that is commonly overlooked here is that the RSS feed could contain some malicious JavaScript code and it would then execute with the privileges of the extension—meaning that it would get full access to the browser (cookies, history etc) and to user's files" [sic].

We characterize these cross-protection-domain interactions as information-flow patterns from JavaScript objects that include page content (untrusted sources) to JavaScript objects and methods that execute content with chrome privileges (executable sinks). In this section we discuss the sources and sinks that VEX tracks. Flows between these sources and sinks are sometimes benign, and represent an incomplete list of possible extension security bugs, but these are the patterns that VEX considers suspicious.

4.2.1 Untrusted Sources

We now describe the untrusted JavaScript objects that extensions can access. Untrusted objects might contain foreign scripts that can lead to attacks if run with chrome privileges.

The JavaScript content-document object (`window.content.document`) accesses the browser's content page directly, and hence is an untrusted source. Also, the browser sets JavaScript pop-up nodes (`document.popupNode`) when the user right-clicks on document object model (DOM) elements. If this DOM element is part of the page content, then it includes untrusted page content.

One API that extensions use to access persistent state is the Resource Descrip-

tion Framework (RDF). RDF is a model for describing hierarchical relationships between browser resources [64] and is used by the browser to store persistent data, like bookmarks. Extension developers can store persistent extension data in an RDF file, or access browser resources stored in RDF format. However, RDF data can come from untrusted sources. For example, when a user stores a bookmark, Firefox records the un-sanitized title of the bookmarked page, which is controlled by the web page, in an RDF file. Extensions can also access un-sanitized bookmark URLs using the `nsILivemarkService` interface and the `BookmarksUtils` object.

Extensions access Firefox preferences through the `nsIPrefService` interface. Any extension can set values in the preferences, and extensions have unchecked access to all preference settings. Some extensions use this service to store untrusted strings obtained from web page content; hence using this service is also treated as an untrusted source.

In summary, the VEX treats the following as untrusted source objects: `window.content.document`, `document.popupNode`, `BookmarksUtils`, and access to the new instances of the objects `nsIRDFService`, `nsILivemarkService`, and `nsIPrefService`.

4.2.2 Executable Sinks

Now we describe the set of executable sinks, which are JavaScript objects and methods that provide a way to parse and execute JavaScript dynamically. VEX considers these executable sinks to be potentially dangerous when they execute untrusted JavaScript code with chrome privileges.

The `eval` function call interprets string data as JavaScript, which it executes dynamically. This flexible mechanism can be used to generate JavaScript code dynamically, for example to deserialize JavaScript Object Notation (JSON) objects. However, this flexibility can lead to code injection vulnerabilities in extensions. If extensions execute `eval` functions on un-sanitized strings that come from untrusted sources, an attacker can inject JavaScript code that runs with full chrome privileges.

Each HTML element in a page has an `innerHTML` property that defines the text that occurs between that element's opening and closing tags. Extensions can change the `innerHTML` property to alter existing DOM elements, or to add new

DOM elements, because the browser parses the modified text after JavaScript code modifies this property. Thus, passing specially crafted strings (e.g., <img> tags with JavaScript in their `onload` attribute) into `innerHTML` can lead to code injection attacks.

Extensions can add a new DOM element to a content page or chrome page by using the `appendChild` method. This method causes the browser to parse and process the data within the element, similar to the `innerHTML` property. Therefore, this feature can also be used to execute injected code.

In summary, the executable sinks that we consider in VEX are calls to the functions `eval` and `appendChild`, and assignments to `innerHTML` property.

4.3 Static Information Flow Analysis of JavaScript

The core component of VEX is a static analysis tool for detecting explicit information flows in browser extensions written in JavaScript. VEX computes flows between different sources and sinks, including all those described in Section 4.2. To support fine-grained information-flow analysis, VEX tracks the flows from source objects to the sinks encountered in the JavaScript extension, using a taint-based analysis. Motivated by the fact that every flow reported needs to be checked manually for attacks, which can take considerable human effort,we aim for an analysis that admits as few false positives as possible, where false positives are flows reported by VEX that cannot actually occur at run time.

Statically analyzing JavaScript extensions for flows is a non-trivial task. Object properties in JavaScript change dynamically, in the sense that new object properties can be created dynamically at run-time. Functions are objects in JavaScript, and hence can be created, redefined dynamically, and passed as parameters. In addition to the objects defined in the program, the extensions can also access the browser's DOM API and the Firefox Extension API provided by XPCOM components, and the static analysis must handle them correctly. JavaScript browser extensions also have a large number of objects and functions that need to be tracked. The challenge is to accurately keep track of such objects, properties, and the corresponding flows to them.

The analysis engine in VEX is a static taint analysis to detect explicit flows, where taint propagation for JavaScript is achieved by adapting an operational semantics for JavaScript proposed by Maffeis *et al* [39]. In the analysis, we replace

concrete heaps by *abstract heaps*, where abstract heaps accurately track objects and their properties, but abstract the primitive values. An abstract heap can be seen as a directed graph, where every object and function in the JavaScript program is represented as a node, while the edges in the heap represent the field relationships between different objects. Additionally, every node in the abstract heap is associated with a taint value, which is used by VEX's analysis to compute the information flows from the source objects to the sinks.

The Analysis: In the analysis, VEX handles only loop-free programs, and translates programs with loops to loop-free programs first by unrolling loops a bounded number of times (hence the analysis is not *sound*– see Section 4.3.6). The VEX abstract semantics computes and tracks the abstract heap on (loop-free) programs fairly precisely by mimicking the operational semantics for JavaScript. Unlike common abstraction domains used in the literature, at any point during the analysis, an abstract heap does not have a single node representing two objects; hence VEX is quite accurate in keeping track of the precise heap nodes and field relations and the corresponding flows, ignoring only the exact primitive values in the heap (like integers). Since programs are unrolled into loop-free code, the abstract heaps have a bounded size, leading to a terminating algorithm.

JavaScript core objects and functions are summarized to have only the essential functionality; an example summary is given in Section 4.3.5. Variables and functions that are not initialized in the current program execution or through summaries, are initialized to point to placeholder dummy objects with HIGH taints. The default taint of an object created in the extension is set to LOW unless the analysis explicitly sets the value to HIGH or a variable is uninitialized. The loops in the program are unrolled a bounded number of times and function calls are inlined for a bounded unrolling of recursive calls, and every path of the resulting program is explored. Thus VEX may overlook certain flows, as discussed in Section 4.3.6. The static analysis does not evaluate the conditions in conditional statements of the program because of the abstraction. Whenever it reaches a conditional statement, both branches are traversed, in a depth-first manner, to ensure that the entire program is covered. The analysis is flow-sensitive and, due to inlining, also context-sensitive.

In Section 4.3.1, we list the core JavaScript syntax and the JavaScript program values. In section 4.3.2, we describe the abstract heap in detail, followed by a description of the data structures that we use for the static analysis. The core

syntax is used for defining the abstract operational semantics in Section 4.3.4.
Notation: In the following sections, the components enclosed in square brackets ([ ]) are optional and the components enclosed in quotes (“ ”) are required. A finite list of elements of a particular type is represented by the appropriate non-terminal symbol followed by a tilde(˜).

4.3.1 Core JavaScript Syntax

$\texttt{val}^{js}$ ::=	$\texttt{pv}^{js} \mid \texttt{fd}^{js}$	% *values*
$\texttt{pv}^{js}$ ::=	$\texttt{c}^{js} \mid \texttt{undefined}$	% *primitive value*
$\texttt{c}^{js}$ ::=	$\texttt{m}^{js} \mid \texttt{n}^{js} \mid \texttt{bool}^{js} \mid \texttt{null}$	% *literals*
$\texttt{m}^{js}$ ::=	“foo” | “bar” | ...	% *strings*
$\texttt{n}^{js}$ ::=	$-\texttt{n} \mid 0 \mid 1 \mid \ldots$	% *numbers*
$\texttt{bool}^{js}$ ::=	$\texttt{true} \mid \texttt{false}$	% *boolean values*
id ::=	$foo \mid bar \mid \ldots$	% *identifiers*
$\texttt{i}^{js}$ ::=	$\texttt{m}^{js} \mid \texttt{id} \mid \texttt{n}^{js}$	% *property name*
$\texttt{nonstr}^{js}$ ::=	$\texttt{n}^{js} \mid \texttt{bool}^{js}$	% *non − string literals*
$\texttt{nulllit}^{js}$ ::=	$\texttt{null} \mid \texttt{undefined}$	% *null literals*
$\texttt{fd}^{js}$ ::=	$\texttt{function}$“(”$[\texttt{id}_1, \ldots, \texttt{id}_k]$“)”{”$[prog]$“}”	% *function decl*

Figure 4.2: JavaScript Values

Figure 4.2 gives the values that can be used in the JavaScript program. In this figure, we add a superscript js to the non-terminals to indicate that these symbols belong to the core JavaScript syntax; in Figure 4.4 we present the values used in our analysis. Apart from the usual strings, booleans, and numbers, JavaScript has some additional primitive values – `null` is an object that is used to represent the objects that do not exist (i.e. can't find them anywhere in the scope chain), and `undefined` is a value used to indicate that the variable has been defined but not been assigned a value. The JavaScript property names could be either strings, identifiers, or numbers.

Figure 4.3 gives the syntax of the core components of the JavaScript programming language. A core JavaScript expression could be a primitive value, object or array literal, an identifier, a special `this` object, an expression generated using different operators, a function definition, a function or a constructor call, etc. The JavaScript programming language statements are similar to the statements in other programming languages which support exceptions. A JavaScript programming language is a sequence of program statements or function definitions. VEX

analysis consists of the operational semantics for analyzing the JavaScript core syntax.

	EXPRESSIONS		
expr	::=	$\texttt{pv}^{js}$	(PRIMITIVE VALUE)
	|	"{" [$\texttt{i}_1^{js}$: *expr*$_1$, ..., $\texttt{i}_n^{js}$: *expr*$_n$] "}"	(OBJECT LITERAL)
	|	"[" [*expr*$_1$, ..., *expr*$_n$] "]"	(ARRAY LITERAL)
	|	`this`	(THIS)
	|	`id`	(IDENTIFIER)
	|	*expr*.`id`	(FIELD ACCESS)
	|	*expr* "[" *expr* "]"	(MEMBER SELECTOR)
	|	*expr* &`BIN` *expr*	(BINARY OP)
	|	&`UN` *expr*	(UNARY OP)
	|	*expr* &`PO`	(POSTFIX OP)
	|	`function` [$\texttt{id}^{js}$] "(" [$\texttt{id}_1^{js}$, ..., $\texttt{id}_n^{js}$] ")" "{" [*prog*] "}"	([NAMED] FUNCTION DEF)
	|	*expr* "(" [*expr*$_1$, ..., *expr*$_n$] ")"	(FUNCTION CALL)
	|	`new` *expr* "(" [*expr*$_1$, ..., *expr*$_n$] ")"	(CONSTRUCTOR CALL)
	|	"(" *expr* "?" *expr* " : " *expr* ")"	(CONDITIONAL EXPRESSION)
	STATEMENTS		
stm	::=	;	(SKIP)
	|	"{" *stm** "}"	(BLOCK)
	|	`var` [(`id` [" = " *expr*]) ~]	(VARIABLE DECL.)
	|	`id` " = " *expr*	(ASSIGN 1)
	|	*expr*.`id` " = " *expr*	(ASSIGN 2)
	|	*expr* "[" *expr* "]" " = " *expr*	(ASSIGN 3)
	|	`if` "(" *expr* ")" *stm*$_1$ [`else` *stm*$_2$]	(CONDITIONAL)
	|	`while` "(" *expr* ")" *stm*	(WHILE)
	|	`return` *expr*	(RETURN)
	|	`break` [`id`]	(BREAK)
	|	`continue` [`id`]	(CONTINUE)
	|	`id` : *stm*	(LABEL)
	|	`throw` *expr*	(THROW)
	|	`try` "{" *stm** "}" [`catch` "(" `id` ")" "{" *stm*$_1^*$ "}"] [`finally` "{" *stm*$_2^*$ "}"]	(TRY)
	PROGRAM		
prog	::=	*stm* [*prog*]	(STATEMENT)
	|	`function` `id` $''$("[$\texttt{id}_1$, ..., $\texttt{id}_n$]$''$) {"[*prog*]"}"	(FUNCTION DECLARATION)

Figure 4.3: Core JavaScript Syntax.

4.3.2 Abstract Heaps

$\mathtt{H} ::=$	$\{(\mathtt{l}_1 : \mathtt{o}_1), (\mathtt{l}_2 : \mathtt{o}_2), \ldots, (\mathtt{l}_n : \mathtt{o}_n)\}$	$\%$ *heap*
$\mathtt{l} ::=$	$\sharp\mathtt{id} \mid \mathtt{rand}$	$\%$ *location*
$\mathtt{id} ::=$	$foo \mid bar \mid \ldots$	$\%$ *identifiers*
$\mathtt{o} ::=$	$\text{“}\{\text{”}[(\mathtt{i}_1 : \mathtt{ov}_1), \ldots, (\mathtt{i}_m : \mathtt{ov}_m)]\text{“}\}\text{”}$	$\%$ *objects*
$\mathtt{i} ::=$	$\mathtt{m}$	$\%$ *property name*
$\mathtt{ov} ::=$	$\mathtt{ln} \mid \mathtt{sectype} \mid \mathtt{m}$	$\%$ *object values*
$\mathtt{sectype} ::=$	$\langle \mathtt{m}, \mathtt{taint} \rangle$	$\%$ *security type*
$\mathtt{m} ::=$	$\mathtt{m}^{js}$	$\%$ *strings*
$\mathtt{taint} ::=$	LOW | HIGH	$\%$ *taint*
$\mathtt{ln} ::=$	$\mathtt{l} \mid$ NULL	$\%$ *nullable addresses*
$\mathtt{r} ::=$	$\mathtt{ln}\text{“}*\text{”}\mathtt{m} \mid$ NULL	$\%$ *references*
$\mathtt{term} ::=$	$\&\mathtt{Normal} \mid \&\mathtt{Return}$	$\%$ *return label*
$\mathtt{vexbool} ::=$	$\&\mathtt{false} \mid \&\mathtt{true}$	$\%$*boolean analysis values*

Figure 4.4: Abstract Heap, Locations and Values

Figure 4.4 gives the abstract data structure that the VEX static analysis tracks, along with the values used in the abstract data structure. We model the state of a JavaScript program using the notion of an abstract heap. Every object is stored on the heap. The heap, `H`, is modeled as a set of (location, object) pairs. A location, `l`, is an arbitrarily generated name created whenever a new object is created in the program. For readability, a location referenced by an identifier, `id`, is represented as $\sharp$`id`. An object, `o`, represents the JavaScript object data structure and is a set of (abstract property name, abstract value) pairs. An abstract property name, `i`, is a primitive string. An abstract value, `ov`, could be a heap location (if the property points to another object or if it is a primitive value), a NULL object, a string (value of a string in the program syntax or a function body), or a security type. Only the string primitive values are preserved in the String object. Security type, `sectype`, keeps track of taints; a sink object's security type acquires a taint associated with a source object, if there is an explicit flow from the source object to the sink object. A security type is modeled as a pair ⟨taint value, source string⟩; the taint value, `taint`, could either be LOW or HIGH and the taint source string, `m`, is a string identifying the source object of the taint. A reference, `r`, is an intermediate value used to represent the property accesses of various objects in the program. The return label, `term`, is used to represent different states of the analysis as described in Section 4.3.4.

4.3.3 Abstraction Function

VEX analysis abstracts out certain primitive values in order to provide tractable static analysis of JavaScript programs. The abstraction function maps the primitive values, property names, and literals in the original JavaScript program (given in figures 4.2 and 4.3) into abstract heap values (given in figure 4.4). The mapping is as follows:

- The $\texttt{nulllit}^{js}$ values are mapped to NULL.
- The primitive integers ($\texttt{n}^{js}$) and booleans ($\texttt{bool}^{js}$) are mapped into objects created using the new_primitive function. However, the exact values of these literals are not stored in the object.
- The primitive strings ($\texttt{m}^{js}$) are mapped into objects created using the new_string_primitive function. This string object records the exact string primitive value for reference in the analysis.
- An object literal ($\{[(\texttt{i}_1^{js} : \mathit{expr}_1), \ldots, (\texttt{i}_n^{js} : \mathit{expr}_n)]\}$), is mapped into an object representation which faithfully keeps track of the abstraction of the property and value mappings. During the abstract object creation, the integer and identifier property names are converted into primitive strings by the operational semantics.
- An array literal is also mapped onto an abstract object. Special properties are added to this object by creating the object as an instance of the predefined Array object.
- A function declaration is also mapped into an abstract object with a special property, "@Body", which stores the function body as a string.
- For objects that are dynamically created inside loops, our operational semantics faithfully maps the exact object representation for a pre-specified number of loop iterations and then ignores the rest of heap once the maximum number of iterations is reached.

In summary, every element in the concrete heap is mapped into an abstract object. Along with the abstract heap value, the analysis computes and propagates a taint value with every heap object. The object creation and taint propagation algorithm is given by the abstraction operational semantics in Section 4.3.4

4.3.4 Abstract Operational Semantics

Operational semantics of a language give a set of rules for evaluating the program, its statements and its expressions to produce a state, given an initial mapping of variables to values in the form of an initial state. We model VEX's program state, σ, is either a tuple $\langle$*current heap*, *current location*, *global location*, *program fragment*$\rangle$ ($\sigma = \langle \mathtt{H}, \mathtt{l}_s, \mathtt{l}_\mathtt{g}, \mathtt{fragment} \rangle$) or a tuple $\langle$*current heap*, *current location*, *global location*, *program context*$\rangle$ ($\sigma = \langle \mathtt{H}, \mathtt{l}_s, \mathtt{l}_\mathtt{g}, \mathtt{co} \rangle$)

- An abstract heap object, H, is a set of (location,object) pairs representing the objects that are initialized and used in the JavaScript program.
- A current scope object, $\mathtt{l}_s$, is a pointer to a location in the heap that represents the current execution context. All the local variables are accessible from the current scope.
- A global object, $\mathtt{l}_\mathtt{g}$, is a pointer to the initial object in the heap that consists of all the core JavaScript objects like `Object`, `Array`, `Function`, etc. Initially, the current scope object is set to the location pointing to the global object.
- A program fragment, `fragment`, is either a *prog*, *stm*, or *expr*.
- A program context, `co`, is a tuple $\langle$ *return label*, *object reference*, *return value* $\rangle$ ($\langle \mathtt{term}, \mathtt{r}, \mathtt{ov} \rangle$). The value of the return label, `term`, could be one of `&Normal` or `&Return`, representing the termination status of the program constructs. The return value, `ov`, gives the value stored in the location. The object reference, `r`, is the reference to the returned value that is obtained by evaluating the statement or the expression.

VEX analysis is based on a set of rules that transform abstract heaps based on each statement in the program. The analysis works by over-approximating the effect of the statements on the abstract heap. In this section, we present the big-step operational semantic rules for expressions, statements, and programs. The transformation rules are denoted respectively by $\xrightarrow{expr}$, $\xrightarrow{stm}$, and $\xrightarrow{prog}$ respectively. Given a state containing a program fragment, the transformation rules produce a state containing new program context after their execution.

These rules closely follow the small-step operational semantics proposed by Maffeis *et al* [39], which cover the ECMA-262 standard for JavaScript. The heap

representation abstracts away the primitive values and the types. Therefore, big-step operational semantics are sufficient for the analysis since it does not require fine-grained control over the state and the computations. Whenever applicable, we point out the differences between our abstract semantics and those proposed by Maffeis *et al*.

Program Semantics

Programs are sequence of statements and function declarations. The program semantics are defined in Figure 4.5.

$$\frac{\text{VD}(\texttt{NativeEnv}, \texttt{l}_\texttt{g}, \texttt{l}_\texttt{g}, P) = \texttt{H}_1 \quad \text{FD}(\texttt{H}_1, \texttt{l}_\texttt{g}, \texttt{l}_\texttt{g}, P) = \texttt{H}_2}{\mathit{prog} \xrightarrow{\mathit{prog}} \texttt{H}_2, \texttt{l}_\texttt{g}, \texttt{l}_\texttt{g}, \mathit{prog}} \text{ (P-INIT)}$$

$$\texttt{H}, \texttt{l}, \texttt{l}_\texttt{g}, \texttt{function id}([\texttt{id} \sim])\{[\mathit{prog}]\}[\mathit{prog}_1] \xrightarrow{\mathit{prog}} \texttt{H}, \texttt{l}, \texttt{l}_\texttt{g}, [P1] \quad \text{(P-FUN)}$$

$$\frac{\begin{array}{l}\texttt{H}, \texttt{l}, \texttt{l}_\texttt{g}, \mathit{stm} \xrightarrow{\mathit{stm}} \texttt{H}', \texttt{l}', \texttt{l}_\texttt{g}, \langle \&\texttt{Normal}, \texttt{r}_1, \texttt{ov}_2 \rangle \\ \texttt{H}', \texttt{l}', \texttt{l}_\texttt{g}, \mathit{prog} \xrightarrow{\mathit{prog}} \texttt{H}'', \texttt{l}'', \texttt{l}_\texttt{g}, \langle \texttt{term}, \texttt{r}'_1, \texttt{ov}'_2 \rangle\end{array}}{\texttt{H}, \texttt{l}, \texttt{l}_\texttt{g}, \mathit{stm}; \mathit{prog} \xrightarrow{\mathit{prog}} \texttt{H}'', \texttt{l}'', \texttt{l}_\texttt{g}, \langle \texttt{term}, \texttt{r}'_1, \texttt{ov}'_2 \rangle} \text{ (P-SEQ)}$$

$$\texttt{H}, \texttt{l}, \texttt{l}_\texttt{g}, \xrightarrow{\mathit{prog}} \texttt{H} \qquad \text{(P-TERM-BRANCH)}$$

Figure 4.5: Program Semantics

Given a program *prog*, the rule (P-INIT) initializes the analysis by generating the initial program state. The initial scope points to the global location. This rule starts with an initial heap, `NativeEnv`, which consists of the JavaScript core objects, and the DOM and Firefox API. In JavaScript, function declarations and variable declarations are processed before the program starts executing. The auxiliary functions FD and VD (defined in Figure 4.6) are used to perform the preprocessing in the initialization rule. For every variable declaration, VD creates an entry in the current scope object with the variable name and assigns a NULL value to it. For every function declaration, FD creates an entry for that function in the current scope.

After the pre-processing step, a function declaration encountered during the analysis is ignored as specified in the rule (P-FUN). Blocks of statements are analyzed sequentially using the (P-SEQ) rules.

$$\mathrm{VD}(\mathtt{H}, \mathtt{l},) = \mathtt{H} \quad \text{(H-VD)}$$

$$\frac{\text{“}\mathtt{id}\text{”} \not< \mathtt{H}(\mathtt{l}) \quad \mathtt{H}(\mathtt{l}.\text{“}\mathtt{id}\text{”} = \textsc{Null}) = \mathtt{H}_1}{\begin{array}{c}\mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathtt{var}\ \mathtt{id}[= \mathit{expr}][, (\mathtt{id}_1[= \mathit{expr}_1])\sim][\mathit{prog}]) = \\ \mathrm{VD}(\mathtt{H}_1, \mathtt{l}, \mathtt{var}\ [(\mathtt{id}_1[= \mathit{expr}_1])\sim][\mathit{prog}])\end{array}} \text{(H-VD-INIT)}$$

$$\frac{\text{“}\mathtt{id}\text{”} < \mathtt{H}(\mathtt{l})}{\begin{array}{c}\mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathtt{var}\ \mathtt{id}[= \mathit{expr}][, (\mathtt{id}_1[= \mathit{expr}_1])\sim][\mathit{prog}]) = \\ \mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathtt{var}\ [(\mathtt{id}_1[= \mathit{expr}_1])\sim][\mathit{prog}])\end{array}} \text{(H-VD-INIT-IGNORE)}$$

$$\mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathtt{var}[\mathit{prog}]) = \mathrm{VD}(\mathtt{H}, \mathtt{l}, [\mathit{prog}]) \quad \text{(H-VD-VAR)}$$

$$\mathrm{VD}(\mathtt{H}, \mathtt{l}, \{\mathit{stm}^*\}[\mathit{prog}]) = \mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathit{stm}^*[\mathit{prog}]) \quad \text{(H-VD-BLOCK)}$$

$$\begin{array}{l}\mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathtt{if}\ \text{“(”}\mathit{expr}\text{“)”}\ \mathit{stm}_1\ \mathtt{else}\ \mathit{stm}_2[\mathit{prog}]) \\ \qquad = \mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathit{stm}_1\ \mathit{stm}_2\ [\mathit{prog}]) \quad \text{(H-VD-IF)}\end{array}$$

$$\mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathtt{while}\ \text{“(”}\mathit{expr}\text{“)”}\ \mathit{stm}[\mathit{prog}]) = \mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathit{stm}\ [\mathit{prog}]) \quad \text{(H-VD-WHILE)}$$

$$\begin{array}{l}\mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathtt{try}\text{“\{”}\mathit{stm}^*\text{“\}”}\mathtt{catch}\text{“(”}\mathtt{id}\text{“)”}\text{“\{”}\mathit{stm}_1^*\text{“\}”}\mathtt{finally}\text{“\{”}\mathit{stm}_2^*\text{“\}”}[\mathit{prog}]) \\ \qquad = \mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathit{stm}^*\ \mathit{stm}_1^*\ \mathit{stm}_2^*\ [\mathit{prog}]) \quad \text{(H-VD-TRY)}\end{array}$$

$$\frac{\begin{array}{l}\mathit{stm} < \{\text{“;”}, \mathit{expr}, \text{“}\mathtt{continue}\ [\mathtt{id}];\text{”}, \text{“}\mathtt{break}\ [\mathtt{id}];\text{”}, \\ \quad \text{“}\mathtt{return}\ \ \mathit{expr};\text{”}, \text{“}\mathtt{throw}\ \ \mathit{expr}\text{”}\}\end{array}}{\mathrm{VD}(\mathtt{H}, \mathtt{l}, \mathit{stm}[\mathit{prog}]) = VD(\mathtt{H}, \mathtt{l}, [\mathit{prog}])} \text{(H-VD-IGNORE)}$$

$$\mathrm{FD}(\mathtt{H}, \mathtt{l},) = \mathtt{H} \quad \text{(H-FD)}$$

$$\frac{\begin{array}{c}\mathtt{H}, \mathrm{Function}(\mathtt{function}\ ([\mathtt{id}\sim])\{;[\mathit{prog}]\}, \mathtt{l}) = \mathtt{H}_1, \mathtt{l}_1 \\ \mathtt{H}_1(\mathtt{l}.\text{“}\mathtt{id}_{fun}\text{”} = \mathtt{l}_1) = \mathtt{H}_2\end{array}}{\begin{array}{c}\mathrm{FD}(\mathtt{H}, \mathtt{l}, \mathtt{function}\ \mathtt{id}_{fun}([\mathtt{id}\sim])\{[\mathit{prog}]\}[\mathit{prog}_1]) \\ = \mathrm{FD}(\mathtt{H}_2, \mathtt{l}, [\mathit{prog}_1])\end{array}} \text{(H-FD-FD)}$$

$$\frac{\mathit{stm} < \{\mathtt{while}, \mathtt{for}, \ldots \mathtt{function}\}}{\mathrm{FD}(\mathtt{H}, \mathtt{l}, \mathit{stm}[\mathit{prog}]) = \mathrm{FD}(\mathtt{H}, \mathtt{l}, [\mathit{prog}])} \text{(H-FD-IGNORE)}$$

Figure 4.6: Variable Instantiation and Function Declaration

Statement Semantics

The semantics of statements are defined in Figure 4.7. The ; statement does not change the program state. The return program state of an expression statement is the same as the program state obtained by evaluating the expression; this is defined in the rule (S-EXPR). The conditional statement without an else branch is transformed into a statement with an else branch consisting only of the skip statement in rule (S-IF). The analysis parallelly evaluates every branch in the program, by evaluating both the branches of the conditional statements. The rules (S-IF-THEN) and (S-IF-ELSE) illustrate this. The analysis does not evaluate the conditional expression. This is one of the main differences between the operational semantics of JavaScript and out static semantics.

Similarly, for the while statement the analysis should be able to skip the while loop and continue the analysis on the rest of the program, or it should execute the body of the loop a pre-specified number of times. The (S-WHILE-SKIP) rule shows the skipping of the while statement. The (S-WHILE-BODY) rule shows that the body of the loop is executed and the resulting state is used to execute the next iteration of the while loop. Therefore, the result of executing the while loop is the state obtained by performing a few iterations over the loop body. The loop conditional is also ignored by the analysis. The loop is executed a fixed number of times instead. This is another difference between the VEX analysis and the operational semantics.

The assignment statement is defined in the rule (S-ASSIGN). The left hand side and the right hand side expressions are evaluated using the expression semantics given in Section 4.3.4. The expression semantic rules return the program context $\langle \&\texttt{Normal}, \texttt{l}_1 * \texttt{m}, \texttt{ov}_1 \rangle$ for the left hand side expression $expr_1$ and the program context $\langle \&\texttt{Normal}, \texttt{r}_2, \texttt{ov}_2 \rangle$ for the right hand side expression $expr_2$. The (S-ASSIGN) assigns the value obtained from the right hand side, $\texttt{ov}_2$, to the reference of the left hand side, $\texttt{l}_1 * \texttt{m}$, using the @Put auxiliary function defined in the Figure 4.20.

Expression Semantics

Given a program state, σ, an expression returns a new program state along with the program context, which contains the reference to the location pointed by the expression and the value stored at that location. We describe the expression se-

%Skip Statement

$$\mathtt{H}, \mathtt{l}, \mathtt{l_g}, ; \xrightarrow{stm} \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \textsc{Null}, \textsc{Null} \rangle \quad \text{(S-Skip)}$$

%Expression

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr} \xrightarrow{expr} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \mathtt{term}, \mathtt{r}_1, \mathtt{ov}_2 \rangle}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr} \xrightarrow{stm} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \mathtt{term}, \mathtt{r}_1, \mathtt{ov}_2 \rangle} \text{(S-Expr)}$$

%Conditional Statements

%If then else

$$\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{if}(\mathit{expr})\ \mathit{stm}_1 \xrightarrow{stm} \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \textsc{Null}, \textsc{Null} \rangle \quad \text{(S-If-Skip)}$$

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{stm}_1 \xrightarrow{stm} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \mathtt{term}, \mathtt{r}_1, \mathtt{ov}_2 \rangle}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{if}(\mathit{expr})\ \mathit{stm}_1 \xrightarrow{stm} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \mathtt{term}, \mathtt{r}_1, \mathtt{ov}_2 \rangle} \text{(S-If)}$$

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{stm}_1 \xrightarrow{stm} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \mathtt{term}, \mathtt{r}_1, \mathtt{ov}_2 \rangle}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{if}(\mathit{expr})\ \mathit{stm}_1\ \mathtt{else}\ \mathit{stm}_2 \xrightarrow{stm} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \mathtt{term}, \mathtt{r}_1, \mathtt{ov}_2 \rangle} \text{(S-If-then)}$$

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{stm}_2 \xrightarrow{stm} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \mathtt{term}, \mathtt{r}_1, \mathtt{ov}_2 \rangle}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{if}(\mathit{expr})\ \mathit{stm}_1\ \mathtt{else}\ \mathit{stm}_2 \xrightarrow{stm} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \mathtt{term}, \mathtt{r}_1, \mathtt{ov}_2 \rangle} \text{(S-If-else)}$$

%Iteration Statements

%While

$$\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{while}(\mathit{expr})\ \mathit{stm}_1 \xrightarrow{stm} \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \textsc{Null}, \textsc{Null} \rangle \text{(S-While-Skip)}$$

$$\frac{\begin{array}{c}\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{stm}_1 \xrightarrow{stm} \mathtt{H}_1, \mathtt{l}_1, \mathtt{l_g}, \langle \mathtt{term}_1, \mathtt{r}'_1, \mathtt{ov}'_2 \rangle, \\ \mathtt{H}_1, \mathtt{l}_1, \mathtt{l_g}, \mathtt{while}(\mathit{expr})\ \mathit{stm}_1 \xrightarrow{stm} \mathtt{H}_2, \mathtt{l}_2, \mathtt{l_g}, \langle \mathtt{term}_2, \mathtt{r}''_1, \mathtt{ov}''_2 \rangle\end{array}}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{while}(\mathit{expr})\ \mathit{stm}_1 \xrightarrow{stm} \mathtt{H}_2, \mathtt{l}_2, \mathtt{l_g}, \langle \mathtt{term}_2, \mathtt{r}''_1, \mathtt{ov}''_2 \rangle} \text{(S-While-Body)}$$

%var expression and assignment semantics

$$\frac{\begin{array}{c}\mathtt{H}, \mathtt{l}, \mathtt{l_g}, [\mathtt{var}]\ \mathit{expr}_1 \xrightarrow{expr} \langle \&\mathtt{Normal}, \mathtt{l}_1 * \mathtt{m}, \mathtt{ov}_1 \rangle \\ \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}_2 \xrightarrow{expr} \langle \&\mathtt{Normal}, \mathtt{r}_2, \mathtt{ov}_2 \rangle \\ \mathtt{H}, \mathtt{l}, \mathtt{l_g}, @\mathrm{Put}(\mathtt{l}_1 * \mathtt{m}, \mathtt{ov}_2) \rightarrow \mathtt{H'}, \mathtt{l}, \mathtt{l_g}, \mathtt{ov}_2\end{array}}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, [\mathtt{var}]\ \mathit{expr}_1 = \mathit{expr}_2 \xrightarrow{stm} \mathtt{H'}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}_1 * \mathtt{m}, \mathtt{ov}_2 \rangle} \text{(S-Assign)}$$

%Return Statement

$$\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{return}; \xrightarrow{stm} \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Return}, \textsc{Null}, \textsc{Null} \rangle \quad \text{(S-Return)}$$

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr} \xrightarrow{expr} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}, \mathtt{ov} \rangle}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{return}\ \mathit{expr}; \xrightarrow{stm} \mathtt{H'}, \mathtt{l'}, \mathtt{l_g}, \langle \&\mathtt{Return}, \mathtt{r}, \mathtt{ov} \rangle} \text{(S-Return-Expr)}$$

Figure 4.7: Statement Semantics

mantics in this section.

In our abstract semantics, every value is either an object, a string or a security type. The analysis also ignores the exact values of the integer and boolean literals. All boolean operations compute the taints instead of the actual primitive values. This means that the VEX abstract operational semantics need not deal with fine grained type conversion semantics and the contextual rules as in the operational semantics proposed by Maffeis *et al*.

Figure 4.8 defines the expression semantics for JavaScript literals. The rule (E-STRINGLIT) creates a new object for the string using the predefined templates for creating objects defined in Figure 4.24, allocates a new location for this object, and adds the (location, object) pair to the heap, $\mathtt{H}$, to produce the new heap $\mathtt{H}_1$. The new string object keeps track of the value of the string primitive. The rule (E-NON-STRINGLIT) is similar to the previous rule except that it works on the non-string literals and does not track the exact primitive value. The rules (E-OBJLIT), and (E-ARRAYLIT) create new object and array literals respectively. They work by creating a new object for the literals and then inserting the properties into these objects using the @Put internal method defined in Figure 4.20. Both these rules differ in the way the new object is created. In (E-OBJLIT) the new object is created using a predefined template, while in (E-ARRAYLIT) the new array object is created by instantiating an `Array` object (one of the core JavaScript objects). All the literals are tainted LOW.

Figure 4.9 has the semantic rules for `this`, identifier access, field access and member selector. For `this` and `id`, the scope chain is checked to find the scope in which the identifiers could be found. Once the scope object is found, the @Get function is used to get the value of the identifier and the @GetParent function to get the location of the parent. These two functions are defined in Figure 4.20. Notice that the property names in the array object are the array indices converted to strings.

We define the semantics for operator handling in Figure 4.10. In case of unary operators, and postfix operators, the analysis returns the object value that the expression represents as shown in the rules (E-UNARYOP) and (E-POSTFIXOP). Rule (E-BINARYOP) defines the semantics for binary operations on non-string expressions. In this rule, the semantics use a predefined new_object template to create a new object. The security type of the newly created object is the join of the security type of the objects represented by the operands. Additionally, in case of string concatenation (as shown in rule (E-CONCAT)) the analysis concatenates

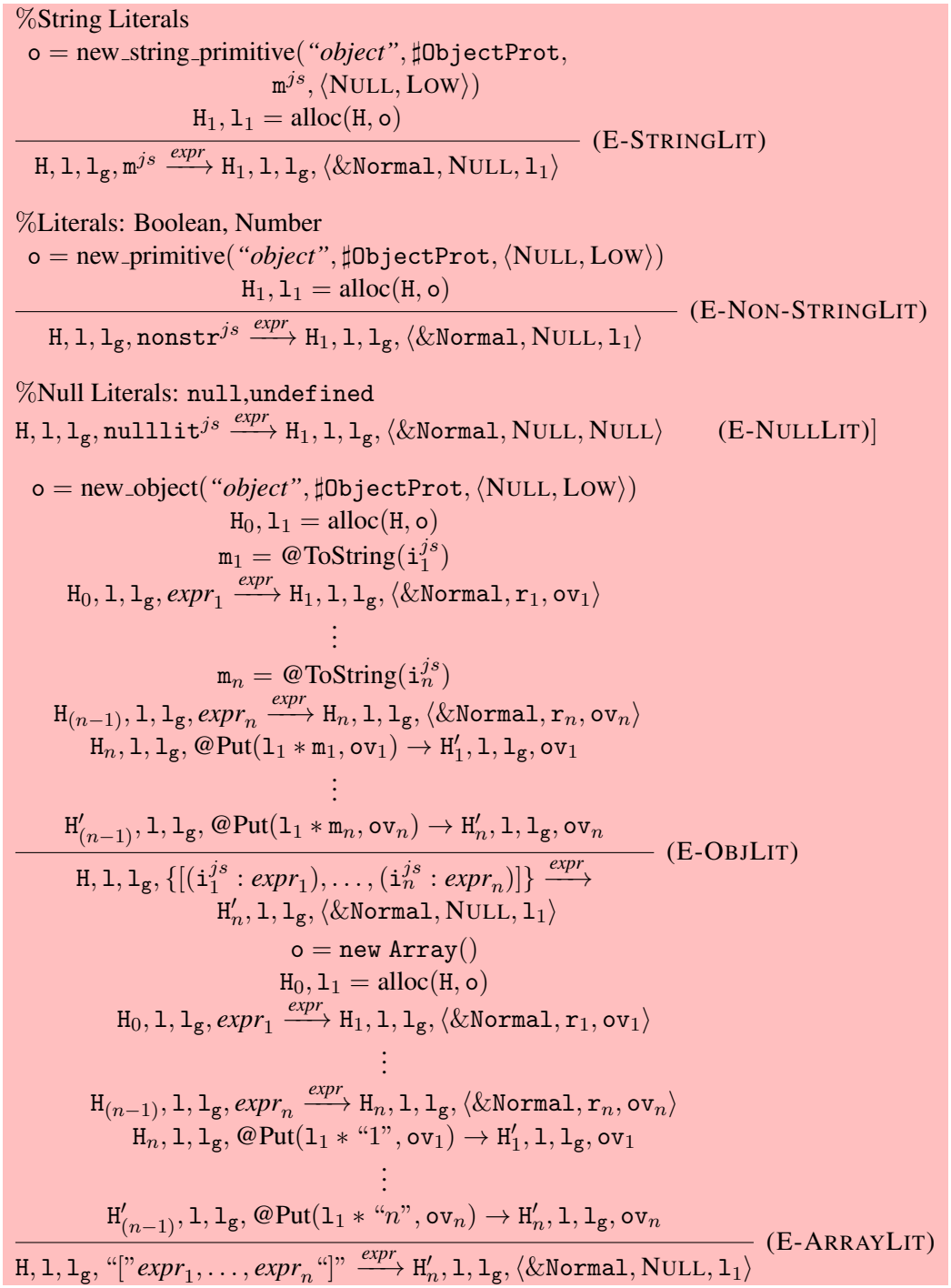

Figure 4.8: Expression Semantics for Literals

$$\frac{\text{Scope}(\mathtt{H}, \mathtt{l}, \text{“}\mathtt{@this}\text{”}) = \mathtt{l}_1 \qquad \mathtt{H}, \mathtt{l}_1.\text{@Get}(\text{“}\mathtt{@this}\text{”}) = \mathtt{ov} \qquad \mathtt{H}, \mathtt{l}_1.\text{@GetParent}(\text{“}\mathtt{@this}\text{”}) = \mathtt{l}_2}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{this} \xrightarrow{expr} \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}_2 * \text{“}\mathtt{@this}\text{”}, \mathtt{ov} \rangle} \text{ (E-THIS)}$$

$$\frac{\text{Scope}(\mathtt{H}, \mathtt{l}, \text{“}\mathtt{id}\text{”}) = \mathtt{l}_1 \qquad \mathtt{H}, \mathtt{l}_1.\text{@Get}(\text{“}\mathtt{id}\text{”}) = \mathtt{ov} \qquad \mathtt{H}, \mathtt{l}_1.\text{@GetParent}(\text{“}\mathtt{id}\text{”}) = \mathtt{l}_2}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{id} \xrightarrow{expr} \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}_2 * \text{“}\mathtt{id}\text{”}, \mathtt{ov} \rangle} \text{ (E-IDE-VAL)}$$

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}[\text{“}\mathtt{id}\text{”}] \xrightarrow{expr} \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}_1 * m, \mathtt{ov} \rangle}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}.\mathtt{id} \xrightarrow{expr} \mathtt{H}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}_1 * m, \mathtt{ov} \rangle} \text{ (E-SEL)}$$

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr} \xrightarrow{expr} \mathtt{H}', \mathtt{l}', \mathtt{l_g}', \langle \&\mathtt{Normal}, \mathtt{r}, \mathtt{l}_1 \rangle \qquad \mathtt{H}', \mathtt{l}_1.\text{@Get}(\mathtt{m}) = \mathtt{ov} \qquad \mathtt{H}', \mathtt{ov}.\text{@AddTaints}(\mathtt{l}_1) \rightarrow \mathtt{H}'', \mathtt{ov}}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}[\mathtt{m}] \xrightarrow{expr} \mathtt{H}'', \mathtt{l}', \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}_1 * m, \mathtt{ov} \rangle} \text{ (E-ACC)}$$

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr} \xrightarrow{expr} \mathtt{H}', \mathtt{l}', \mathtt{l_g}', \langle \&\mathtt{Normal}, \text{NULL}, \text{NULL} \rangle \qquad \mathtt{H}', \mathtt{l_g}.\text{@Get}(\mathtt{m}) = \mathtt{ov} \qquad \mathtt{H}', \mathtt{ov}.\text{@AddTaints}(\mathtt{l}_1) \rightarrow \mathtt{H}'', \mathtt{ov}}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}[\mathtt{m}] \xrightarrow{expr} \mathtt{H}'', \mathtt{l}', \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l_g} * m, \mathtt{ov} \rangle} \text{ (E-ACC-GLOB)}$$

$$\frac{\begin{array}{c}\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr} \xrightarrow{expr} \mathtt{H}', \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}' * \mathtt{m}_1, \text{NULL} \rangle \\ \mathtt{o} = \text{new\_object}(\text{“}\mathit{object}\text{”}, \sharp\mathtt{ObjectProt}, \langle \text{NULL}, \text{LOW} \rangle) \\ \mathtt{H}_1, \mathtt{l}_1 = \text{alloc}(\mathtt{H}, \mathtt{o}) \qquad \mathtt{H}_1(\mathtt{l}_1.\text{@Scope} = \mathtt{l}) = \mathtt{H}_2 \\ \mathtt{H}_2, \mathtt{l}, \mathtt{l_g}, \text{@Put}(\mathtt{l}' * \mathtt{m}_1, \mathtt{l}_1) \rightarrow \mathtt{H}_3, \mathtt{l}, \mathtt{l}_1 \\ \mathtt{H}_3, \mathtt{l}_1.\text{@AddTaints}(\mathtt{l}') \rightarrow \mathtt{H}_3', \mathtt{l}_1\end{array}}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}[\mathtt{m}] \xrightarrow{expr} \mathtt{H}_3', \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}_1 * \mathtt{m}, \text{NULL} \rangle} \text{ (E-ACC-NULL)}$$

Figure 4.9: Identifier and Field Access Semantics

%Unary Operators

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr} \xrightarrow{\mathit{expr}} \mathtt{H'}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}, \mathtt{l'} \rangle}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \&\mathtt{UN}\ \mathit{expr} \xrightarrow{\mathit{expr}} \mathtt{H'}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}, \mathtt{l'} \rangle}\ \text{(E-UnaryOp)}$$

%Postfix Operators

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr} \xrightarrow{\mathit{expr}} \mathtt{H'}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}, \mathtt{l'} \rangle}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}\ \&\mathtt{PO} \xrightarrow{\mathit{expr}} \mathtt{H'}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}, \mathtt{l'} \rangle}\ \text{(E-PostfixOp)}$$

%Binary Operators

$$\frac{\begin{array}{c}\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}_1 \xrightarrow{\mathit{expr}} \mathtt{H}_1, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}_1, \mathtt{l}_1 \rangle \\ \mathtt{H}_1, \mathtt{l}, \mathtt{l_g}, \mathit{expr}_2 \xrightarrow{\mathit{expr}} \mathtt{H}_2, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}_2, \mathtt{l}_2 \rangle \\ \neg(\mathtt{H}_2.\text{IsStrPrim}(\mathtt{l}_1) = \&\mathtt{true} \vee \mathtt{H}_2.\text{IsStrPrim}(\mathtt{l}_2) = \&\mathtt{true}) \\ \mathtt{sectype}_1 = \mathtt{H}_2, \mathtt{l}_1.@\text{Get}(\text{“@Taint”}) \\ \mathtt{sectype}_2 = \mathtt{H}_2, \mathtt{l}_2.@\text{Get}(\text{“@Taint”}) \\ \mathtt{sectype} = @\text{JoinTaint}(\mathtt{sectype}_1, \mathtt{sectype}_2) \\ \mathtt{o} = \text{new\_object}(\text{“}\mathit{object}\text{”}, \sharp\mathtt{ObjectProt}, \mathtt{sectype}) \\ \mathtt{H'}, \mathtt{l'} = \text{alloc}(\mathtt{H}, \mathtt{o})\end{array}}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}_1\ \&\mathtt{BIN}\ \mathit{expr}_2 \xrightarrow{\mathit{expr}} \mathtt{H'}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \text{Null}, \mathtt{l'} \rangle}\ \text{(E-BinaryOp)}$$

%String Concatenation

$$\frac{\begin{array}{c}\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}_1 \xrightarrow{\mathit{expr}} \mathtt{H}_1, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}_1, \mathtt{l}_1 \rangle \\ \mathtt{H}_1, \mathtt{l}, \mathtt{l_g}, \mathit{expr}_2 \xrightarrow{\mathit{expr}} \mathtt{H}_2, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{r}_2, \mathtt{l}_2 \rangle \\ \mathtt{H}_2.\text{IsStrPrim}(\mathtt{l}_1) = \&\mathtt{true} \vee \mathtt{H}_2.\text{IsStrPrim}(\mathtt{l}_2) = \&\mathtt{true} \\ \mathtt{m}_1 = @\text{GetStrPrim}(\mathtt{H}_2, \mathtt{l}_1) \\ \mathtt{m}_2 = @\text{GetStrPrim}(\mathtt{H}_2, \mathtt{l}_2) \\ \mathtt{sectype}_1 = \mathtt{H}_2, \mathtt{l}_1.@\text{Get}(\text{“@Taint”}) \\ \mathtt{sectype}_2 = \mathtt{H}_2, \mathtt{l}_2.@\text{Get}(\text{“@Taint”}) \\ \mathtt{sectype} = @\text{JoinTaint}(\mathtt{sectype}_1, \mathtt{sectype}_2) \\ \mathtt{o} = \text{new\_string\_primitive}(\text{“}\mathit{object}\text{”}, \\ \sharp\mathtt{ObjectProt}, (\mathtt{m}_1 + \mathtt{m}_2), \mathtt{sectype}) \\ \mathtt{H'}, \mathtt{l'} = \text{alloc}(\mathtt{H}_2, \mathtt{o})\end{array}}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathit{expr}_1 + \mathit{expr}_2 \xrightarrow{\mathit{expr}} \mathtt{H'}, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \text{Null}, \mathtt{l'} \rangle}\ \text{(E-Concat)}$$

Figure 4.10: Operator Handling

the string value of both the operands and adds its to the primitive string object created using the new_string_primitive predefined function.

$$\%\text{Function declaration expression; returns Object}$$
$$\frac{\mathtt{H}, \text{Function}(\mathtt{function}\ ([\mathtt{id}\sim])\{[;\mathit{prog}]\}, \mathtt{l}) = \mathtt{H}_1, \mathtt{l}_1}{\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{function}\ ([\mathtt{id}\sim])\{[;\mathit{prog}]\} \xrightarrow{\mathit{expr}} \mathtt{H}_1, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \textsc{Null}, \mathtt{l}_1\rangle} \text{(E-FUN)}$$

$$\frac{\begin{array}{c}\mathtt{o} = \text{new\_object}(\text{“}\mathit{Object}\text{”}, \sharp\mathtt{ObjectProt}, \langle \textsc{Null}, \textsc{Low}\rangle) \\ \mathtt{H}_1, \mathtt{l}_1 = \text{alloc}(\mathtt{H}, \mathtt{o}) \\ \mathtt{H}_1(\mathtt{l}_1.@\mathtt{Scope} = \mathtt{l}) = \mathtt{H}_2 \\ \mathtt{H}_2, \text{Function}(\text{fun}([\mathtt{id}\sim])\{;[\mathit{prog}]\}, \mathtt{l}_1) = \mathtt{H}_3, \mathtt{l}_3 \\ \mathtt{H}_3(\mathtt{l}_1.\text{“}\mathtt{id}\text{”} = \mathtt{l}_3) = \mathtt{H}_4\end{array}}{\begin{array}{c}\mathtt{H}, \mathtt{l}, \mathtt{l_g}, \mathtt{function}\ \mathtt{id}([\mathtt{id}\sim])\{[\mathit{prog}]\} \xrightarrow{\mathit{expr}} \\ \mathtt{H}_4, \mathtt{l}, \mathtt{l_g}, \langle \&\mathtt{Normal}, \mathtt{l}_1 * \text{“}\mathtt{id}\text{”}, \mathtt{l}_3\rangle\end{array}} \text{(E-FUN-NAMED)}$$

Figure 4.11: Function Declaration

Figure 4.11 shows the semantics for function declaration. In case of an unnamed function declaration, a new function object is created using the predefined helper function Function defined in the Figure 4.12. This helper function creates a new function object, creates a location for this object in the heap, and then returns this newly created location, $\mathtt{l}_1$. This location is returned by the expression transition in rule (E-FUN). We define the semantics for named function declaration in rule (E-FUN-NAMED). In this rule, a new object, o, is created to store the current scope chain. This new object is assigned to a new location and is inserted into the heap. The new function object is created using the Function method at location $\mathtt{l}_3$. An entry for this new function object is created in the heap and is returned. The Function method is defined by the rule (H-FUNCTION) in Figure 4.12.

$$\%\text{Function object creation}$$
$$\frac{\begin{array}{c}\mathtt{o}_1 = \text{new\_object}(\text{“}\mathit{object}\text{”}, \sharp\mathtt{ObjectProt}, \langle \textsc{Null}, \textsc{Low}\rangle) \\ \mathtt{H}_1, \mathtt{l}_1 = \text{alloc}(\mathtt{H}, \mathtt{o}_1) \\ \mathtt{o} = \text{new\_function}(\mathtt{function}([\mathtt{id}\sim])\{\mathit{prog}\}, \mathtt{l}, \mathtt{l}_1, \langle \textsc{Null}, \textsc{Low}\rangle) \\ \mathtt{H}_2, \mathtt{l}_2 = \text{alloc}(\mathtt{H}_1, \mathtt{o})\mathtt{H}_3 = \mathtt{H}_2(\mathtt{l}_1.\text{“}\mathit{constructor}\text{”} = \mathtt{l}_2)\end{array}}{\mathtt{H}, \text{Function}(\mathtt{function}([\mathtt{id}\sim])\{P\}, \mathtt{l}) = \mathtt{H}_3, \mathtt{l}_2} \text{(H-FUNCTION)}$$

Figure 4.12: Function Object Creation

The function call semantics are given in Figures 4.13, 4.14, 4.15. The semantic rules perform the following operations in order:

%function calls. Returns: Object, Exception. Uses: @Fun, @Call

$$\frac{\begin{array}{c}\mathtt{H},\mathtt{l},\mathtt{l_g},\mathit{expr}_f \xrightarrow{\mathit{expr}} \mathtt{H}',\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{ln}*\mathtt{m},\mathtt{ov}_f\rangle \\ \mathtt{H}',\mathtt{l},\mathtt{l_g},\mathit{expr}_1 \xrightarrow{\mathit{expr}} \mathtt{H}_1,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_1,\mathtt{ov}_1\rangle \\ \vdots \\ \mathtt{H}_{n-1},\mathtt{l},\mathtt{l_g},\mathit{expr}_n \xrightarrow{\mathit{expr}} \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_n,\mathtt{ov}_n\rangle \\ \text{Type}(\mathtt{ln}*\mathtt{m}) = \textsc{Reference} \qquad \text{!isActivation}(\mathtt{H}_n,\mathtt{ln}) \\ \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\text{@Fun}(\mathtt{ln},\mathtt{ln}*\mathtt{m},\mathtt{ov}_1,\ldots,\mathtt{ov}_n) \to \mathtt{H}_{res},\mathtt{l}_{res},\mathtt{l_g},\mathtt{co}\end{array}}{\mathtt{H},\mathtt{l},\mathtt{l_g},\mathit{expr}_f(\mathit{expr}_1,\ldots,\mathit{expr}_n) \xrightarrow{\mathit{expr}} \mathtt{H}_{res},\mathtt{l}_{res},\mathtt{l_g},\mathtt{co}} \text{ (E-CALL-REF)}$$

$$\frac{\begin{array}{c}\mathtt{H},\mathtt{l},\mathtt{l_g},\mathit{expr}_f \xrightarrow{\mathit{expr}} \mathtt{H}',\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{ln}*\mathtt{m},\mathtt{ov}_f\rangle \\ \mathtt{H}',\mathtt{l},\mathtt{l_g},\mathit{expr}_1 \xrightarrow{\mathit{expr}} \mathtt{H}_1,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_1,\mathtt{ov}_1\rangle \\ \vdots \\ \mathtt{H}_{n-1},\mathtt{l},\mathtt{l_g},\mathit{expr}_n \xrightarrow{\mathit{expr}} \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_n,\mathtt{ov}_n\rangle \\ \text{Type}(\mathtt{ln}*\mathtt{m}) = \textsc{Reference} \qquad \text{isActivation}(\mathtt{H}_n,\mathtt{ln}) \\ \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\text{@Fun}(\mathtt{l_g},\mathtt{ln}*\mathtt{m},\mathtt{ov}_1,\ldots,\mathtt{ov}_n) \to \mathtt{H}_{res},\mathtt{l}_{res},\mathtt{l_g},\mathtt{co}\end{array}}{\mathtt{H},\mathtt{l},\mathtt{l_g},\mathit{expr}_f(\mathit{expr}_1,\ldots,\mathit{expr}_n) \xrightarrow{\mathit{expr}} \mathtt{H}_{res},\mathtt{l}_{res},\mathtt{l_g},\mathtt{co}} \text{ (E-CALL-REF-ACT)}$$

$$\frac{\begin{array}{c}\mathtt{H},\mathtt{l},\mathtt{l_g},\mathit{expr}_f \xrightarrow{\mathit{expr}} \mathtt{H}',\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\textsc{Null},\mathtt{ov}_f\rangle \\ \mathtt{H}',\mathtt{l},\mathtt{l_g},\mathit{expr}_1 \xrightarrow{\mathit{expr}} \mathtt{H}_1,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_1,\mathtt{ov}_1\rangle \\ \vdots \\ \mathtt{H}_{n-1},\mathtt{l},\mathtt{l_g},\mathit{expr}_n \xrightarrow{\mathit{expr}} \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_n,\mathtt{ov}_n\rangle \\ \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\text{@Fun}(\mathtt{l_g},\mathtt{ov}_f,\mathtt{ov}_1,\ldots,\mathtt{ov}_n) \to \mathtt{H}_{res},\mathtt{l}_{res},\mathtt{l_g},\mathtt{co}\end{array}}{\mathtt{H},\mathtt{l},\mathtt{l_g},\mathit{expr}_f(\mathit{expr}_1,\ldots,\mathit{expr}_n) \xrightarrow{\mathit{expr}} \mathtt{H}_{res},\mathtt{l}_{res},\mathtt{l_g},\mathtt{co}} \text{ (E-CALL-REF-NULL)}$$

Figure 4.13: Function Call

$$\frac{\begin{array}{c}\text{IsFuncDecl}(\mathtt{H},\mathtt{ov}_1) = \&\mathtt{false} \\ \mathtt{o} = \text{new\_object}(\textit{“dummy”},\mathtt{ObjectProt},\langle \textsc{Null},\textsc{Low}\rangle) \\ \mathtt{H}_2,\mathtt{l}_2 = \text{alloc}(\mathtt{H},\mathtt{o}) \\ \mathtt{H}_2,\mathtt{l}_2.\text{@AddTaints}([\mathtt{ov}\sim]) \to \mathtt{H}'_2,\mathtt{l}_2\end{array}}{\mathtt{H},\mathtt{l},\mathtt{l_g},\text{@Fun}(\mathtt{l}_1,\mathtt{ov}_1[,\mathtt{ov}\sim]) \xrightarrow{\mathit{expr}} \mathtt{H}'_2,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\textsc{Null},\mathtt{l}_2\rangle} \text{ (E-@FUN-EXC)}$$

$$\frac{\begin{array}{c}\text{IsFuncDecl}(\mathtt{H},\mathtt{l}_2) = \&\mathtt{true} \\ \mathtt{H},\mathtt{l},\mathtt{l_g},\mathtt{l}_2.\text{@Call}(\mathtt{l}_1[,\mathtt{ov}\sim]) \to \mathtt{H}',\mathtt{l}',\mathtt{l_g},\mathtt{co}\end{array}}{\mathtt{H},\mathtt{l},\mathtt{l_g},\text{@Fun}(\mathtt{l}_1,\mathtt{l}_2[,\mathtt{ov}\sim]) \xrightarrow{\mathit{expr}} \mathtt{H}',\mathtt{l}',\mathtt{l_g},\mathtt{co}} \text{ (E-@FUN-CALL)}$$

Figure 4.14: @Fun

1. The function expression and all the function arguments are evaluated according to the expression semantic rules.

2. If the function expression evaluates to an program context in which the object reference is a NULL object as in (E-CALL-REF-NULL), then the global location, $\mathtt{l_g}$, is assigned to the `this` property. This condition can occur if the function expression is a function declaration without a name. Additionally, if the parent object of the function object reference, $\mathtt{ln*m}$, is either null or is an activation object, as in the rule (E-CALL-REF-ACT), the global object's location, $\mathtt{l_g}$, is considered to be the `this` property. However, if the parent object obtained from the reference $\mathtt{ln*m}$ is not an activation object, then `ln` is considered to be the `this` property (as in the (E-CALL-REF) rule). All these three rules are illustrated in the Figure 4.13.

3. The helper function @Fun is called with the heap, location of the `this` object, and the function's actual parameters' object values ($\mathtt{ov}_1, \ldots, \mathtt{ov}_n$) if any. This function returns the result value of evaluating the function in the program context `co`, which is returned by the function call expression. The function, @Fun, is defined in the Figure 4.14.

4. The @Fun function takes the function declaration, location of the `this` object, and the function parameter values as arguments along with the current program state as arguments. It produces either a dummy object or the return value of executing the function. A dummy object is created if the function object does not exist $i.e.$ the value returned by evaluating the function name, $\mathtt{ov}_1$, is either null or is not a function declaration object. The taint of the dummy object is the taint of the function parameters. If the value returned by evaluating the function name is a function declaration object, as determined by the IsFuncDecl function, the @Call (defined in Figure 4.15) method is called to execute the function body.

The @Call function essentially inlines the function body and executes the statements in the body. It follows the following steps:

1. Extract the function body, the program statements are in the statement block *prog*.

2. Creates a new arguments object, $\mathtt{o}_{args}$, to store the actual function parameter values. The new_arguments function is defined in the Figure 4.25. It creates

$$\frac{\begin{array}{c}\mathtt{H}(\mathtt{l}_1).@\mathtt{Body} = \mathtt{function}([\mathtt{id} \sim])\{prog\} \\ |\,[\mathtt{ov} \sim]\,| = \mathtt{n} \qquad \text{new\_arguments}(\mathtt{n}, ([\mathtt{ov} \sim]), \mathtt{l}_1, \langle \textsc{Null}, \textsc{Low} \rangle) = \mathtt{o}_{args} \\ \text{alloc}(\mathtt{H}, \mathtt{o}_{args}) = \mathtt{H}_1, \mathtt{l}_3 \qquad \mathtt{H}_1(\mathtt{l}_1).@\mathtt{FScope} = \mathtt{l}_4 \\ \text{new\_activation}(\mathtt{l}_3, \mathtt{l}_2, \mathtt{l}_4) = \mathtt{o}_{act} \\ \text{alloc}(\mathtt{H}_2, \mathtt{o}_{act}) = \mathtt{H}_3, \mathtt{l}_5 \qquad \text{FP}(\mathtt{H}_3, \mathtt{l}_5, ([\mathtt{id} \sim]), 0, n) = \mathtt{H}_4 \\ \text{VD}(\mathtt{H}_4, \mathtt{l}_5, prog) = \mathtt{H}_5 \qquad \text{FD}(\mathtt{H}_5, \mathtt{l}_5, prog) = \mathtt{H}_6 \\ \mathtt{H}_6, \mathtt{l}_5, \mathtt{l}_g, @\text{FunExe}(\mathtt{l}, prog) \rightarrow \mathtt{H}', \mathtt{l}', \mathtt{l}_g, \mathtt{co}\end{array}}{\mathtt{H}, \mathtt{l}, \mathtt{l}_g, \mathtt{l}_1.@\text{Call}(\mathtt{l}_2[, \mathtt{ov} \sim]) \rightarrow \mathtt{H}', \mathtt{l}', \mathtt{l}_g, \mathtt{co}} \text{ (I-CALL)}$$

$$\frac{\mathtt{H}, \mathtt{l}, \mathtt{l}_g, prog \xrightarrow{prog} \mathtt{H}_1, \mathtt{l}_1, \mathtt{l}_g, \mathtt{co} \quad \mathtt{H}, \mathtt{l}_1, \mathtt{l}_g, @\text{FunRet}(\mathtt{l}, \mathtt{co}) \rightarrow \mathtt{H}', \mathtt{l}', \mathtt{l}_g, \mathtt{co}'}{\mathtt{H}, \mathtt{l}, \mathtt{l}_g, @\text{FunExe}(\mathtt{l}, prog) \rightarrow \mathtt{H}', \mathtt{l}', \mathtt{l}_g, \mathtt{co}'} \text{ (I-@FUNEXE)}$$

$$\mathtt{H}, \mathtt{l}, \mathtt{l}_g, @\text{FunRet}(\mathtt{l}_1, \langle \&\mathtt{Return}, \mathtt{r}, \mathtt{ov} \rangle) \rightarrow \mathtt{H}_1, \mathtt{l}_1, \mathtt{l}_g, \langle \&\mathtt{Normal}, \mathtt{r}, \mathtt{ov} \rangle \quad \text{(I-FUN-RET)}$$

$$\mathtt{H}, \mathtt{l}, \mathtt{l}_g, @\text{FunRet}(\mathtt{l}_1, \langle \&\mathtt{Normal}, \mathtt{r}, \mathtt{ov} \rangle) \rightarrow \mathtt{H}, \mathtt{l}_1, \mathtt{l}_g, \langle \&\mathtt{Normal}, \textsc{Null}, \textsc{Null} \rangle \quad \text{(I-FUN-NOR)}$$

%Function parameters

$$\text{FP}(\mathtt{H}, \mathtt{l}, (), \mathtt{n}_1, \mathtt{n}_2) = \mathtt{H} \qquad \text{(H-FP)}$$

$$\frac{\mathtt{n}_1 < \mathtt{n}_2 \qquad \mathtt{l}_1 = \mathtt{H}(\mathtt{l}).\text{“}arguments\text{”} \qquad \mathtt{ov} = \mathtt{H}(\mathtt{l}_1).\text{“}\mathtt{n}_1\text{”} \qquad \mathtt{H}(\mathtt{l}.\text{“}\mathtt{id}\text{”} = \mathtt{ov}) = \mathtt{H}_1}{\text{FP}(\mathtt{H}, \mathtt{l}, (\mathtt{id}[, \mathtt{id} \sim]), \mathtt{n}_1, \mathtt{n}_2) = \text{FP}(\mathtt{H}_1, \mathtt{l}, ([\mathtt{id} \sim]), \mathtt{n}_1 + 1, \mathtt{n}_2)} \text{ (H-FP-ACTUAL)}$$

$$\frac{\mathtt{n}_1 >= \mathtt{n}_2 \quad \mathtt{H}(\mathtt{l}.\text{“}\mathtt{id}\text{”} = \textsc{Null}) = \mathtt{H}_1}{\text{FP}(\mathtt{H}, \mathtt{l}, (\mathtt{id}[, \mathtt{id} \sim]), \mathtt{n}_1, \mathtt{n}_2) = \text{FP}(\mathtt{H}_1, \mathtt{l}, ([\mathtt{id} \sim]), \mathtt{n}_1 + 1, \mathtt{n}_2)} \text{ (H-FP-FORMAL)}$$

Figure 4.15: @Call

an arguments array, where in an entry is created for each object value in the parameters.

3. Extract the function scope associated with the function declaration into location $\mathtt{l}_4$.

4. Create a new activation using the function new_activation, which is also defined in the Figure 4.25.

5. A new activation object, $\mathtt{o}_{act}$, is created and assigned to the location $\mathtt{l}_5$. The activation object consists of three properties, location $\mathtt{l}_3$ pointing to the arguments array, location $\mathtt{l}_2$ pointing to the `this` object, and location $\mathtt{l}_4$ to point to the first location in the scope chain. The activation object is the new scope in which the function is executed.

6. Function FP (defined in Figure 4.15) is used to assign actual parameter values to the formal parameters in the function.

7. Functions VD and FD are used for pre-processing variable and function declarations producing the new heap $\mathtt{H}_6$ and

8. Finally, the previous scope location `l` is stored in the @FunExe context. The function body is executed using the transformation rules and the resulting program context is checked to see if the function returned a value (has the &`Return` label) or it reached the end of the statement block (has the &`Normal` label).

9. If a value is returned, the (I-FUN-RET) rule changes the label to &`Normal`, retains the reference and the returned object value, and returns. The scope is reset to the stored scope.

10. If the function does not return a value, then the (I-FUN-NOR) rule resets the scope, but returns a new context with the reference and the object value set to NULL.

The semantics for constructor invocation are given by the rules in the Figures 4.16, 4.17, and 4.15. The semantics rules are summarized in the following steps:

1. If the constructor expression $expr_f$ does not evaluate to a location as specified in the rule (E-NEW-EXC-OBJ), then a dummy object is created and

%Constructor invocation

% new operator. Returns: Object, Exception . Uses: @Construct

$$\frac{\begin{array}{c}\mathtt{H},\mathtt{l},\mathtt{l},\mathit{expr} \xrightarrow{\mathit{expr}} \mathtt{H}',\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r},\mathtt{ov}\rangle \\ \mathtt{H}',\mathtt{l},\mathtt{l_g},\mathit{expr}_1 \xrightarrow{\mathit{expr}} \mathtt{H}_1,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_1,\mathtt{ov}_1\rangle \\ \vdots \\ \mathtt{H}_{n-1},\mathtt{l},\mathtt{l_g},\mathit{expr}_n \xrightarrow{\mathit{expr}} \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_n,\mathtt{ov}_n\rangle \\ \text{Type}(\mathtt{ov})! = \textsc{Object} \\ \mathtt{ov}_2 = \text{new\_object}(\textit{“object”},\mathtt{ObjectProt},\langle \textit{“dummy”},\textsc{High}\rangle) \\ \mathtt{H}_n,\mathtt{l}_1 = \text{alloc}(\mathtt{H}',\mathtt{o}_2) \\ \mathtt{H}_n,\mathtt{l}_1.@\text{AddTaints}([\mathtt{ov}_1,\ldots,\mathtt{ov}_n]) \rightarrow \mathtt{H}'_n,\mathtt{l}_1\end{array}}{\begin{array}{c}\mathtt{H},\mathtt{l},\mathtt{l_g},\mathtt{new}\ \mathit{expr}[([\mathit{expr}_1,\ldots,\mathit{expr}_n])] \xrightarrow{\mathit{expr}} \\ \mathtt{H}'_n,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\textsc{Null},\mathtt{l}_1\rangle\end{array}}\ \text{(E-New-Exc-obj)}$$

$$\frac{\begin{array}{c}\mathtt{H},\mathtt{l},\mathtt{l},\mathit{expr} \xrightarrow{\mathit{expr}} \mathtt{H}',\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r},\mathtt{l}'\rangle \\ \mathtt{H}',\mathtt{l},\mathtt{l_g},\mathit{expr}_1 \xrightarrow{\mathit{expr}} \mathtt{H}_1,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_1,\mathtt{ov}_1\rangle \\ \vdots \\ \mathtt{H}_{n-1},\mathtt{l},\mathtt{l_g},\mathit{expr}_n \xrightarrow{\mathit{expr}} \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_n,\mathtt{ov}_n\rangle \\ \text{Type}(\mathtt{l}') = \textsc{Object} \qquad @\mathtt{Construct}\ !< \ \mathtt{H}(\mathtt{l}') \\ \mathtt{H}_n,\mathtt{l}'.@\text{AddTaints}([\mathtt{ov}_1,\ldots,\mathtt{ov}_n]) \rightarrow \mathtt{H}'_n,\mathtt{l}'\end{array}}{\begin{array}{c}\mathtt{H},\mathtt{l},\mathtt{l_g},\mathtt{new}\ \mathit{expr}[([\mathit{expr}_1,\ldots,\mathit{expr}_n])] \xrightarrow{\mathit{expr}} \\ \mathtt{H}'_n,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r},\mathtt{l}'\rangle\end{array}}\ \text{(E-New-Exc-constr)}$$

$$\frac{\begin{array}{c}\mathtt{H},\mathtt{l},\mathtt{l},\mathit{expr} \xrightarrow{\mathit{expr}} \mathtt{H}',\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r},\mathtt{l}'\rangle \\ \mathtt{H}',\mathtt{l},\mathtt{l_g},\mathit{expr}_1 \xrightarrow{\mathit{expr}} \mathtt{H}_1,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_1,\mathtt{ov}_1\rangle \\ \vdots \\ \mathtt{H}_{n-1},\mathtt{l},\mathtt{l_g},\mathit{expr}_n \xrightarrow{\mathit{expr}} \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\langle \&\mathtt{Normal},\mathtt{r}_n,\mathtt{ov}_n\rangle \\ \text{Type}(\mathtt{l}') = \textsc{Object} \qquad @\mathtt{Construct}\ < \ \mathtt{H}(\mathtt{l}') \\ \mathtt{H}_n,\mathtt{l},\mathtt{l_g},\mathtt{l}'.@\text{Construct}([\mathtt{ov}_1,\ldots,\mathtt{ov}_n]) \xrightarrow{\mathit{expr}} \mathtt{H}'_n,\mathtt{l}_1,\mathtt{l_g},\mathtt{co}\end{array}}{\mathtt{H},\mathtt{l},\mathtt{l_g},\mathtt{new}\ \mathit{expr}[([\mathit{expr}_1,\ldots,\mathit{expr}_n])] \xrightarrow{\mathit{expr}} \mathtt{H}'_n,\mathtt{l}_1,\mathtt{l_g},\mathtt{co}}\ \text{(E-New-constr)}$$

Figure 4.16: Constructor Invocation

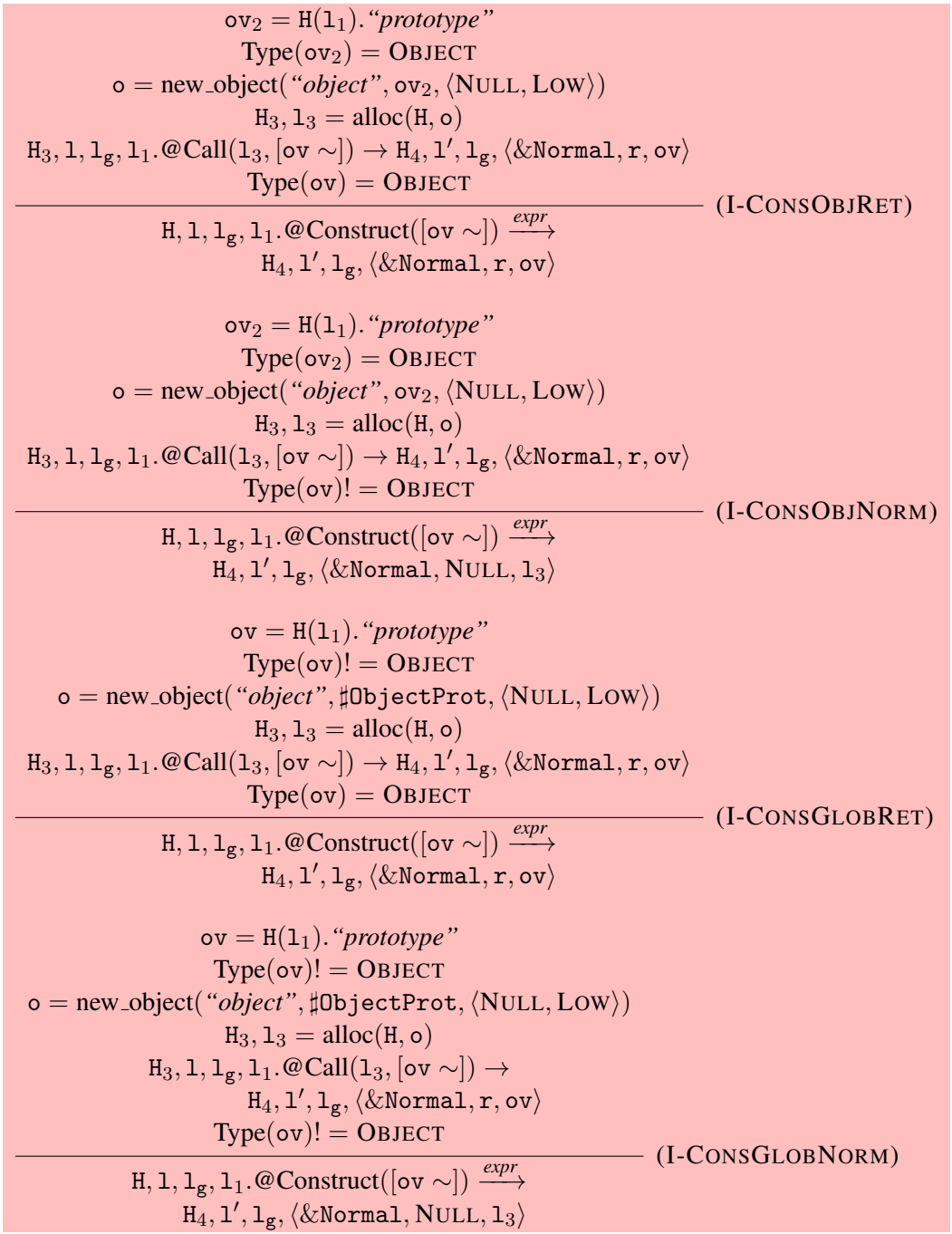

Figure 4.17: @Construct

returned. The dummy object is tainted with the taints of all the parameters to the constructor function.

2. If the constructor expression $expr_f$ evaluates to a location $\mathtt{l}'$, but the object pointed to by this location is not a constructor (i.e. function object), then the location $\mathtt{l}'$, tainted with the security types of the constructor parameters, is returned. This is specified in the rule (E-NEW-EXC-CONSTR).

3. Finally if the expression evaluates to a location $\mathtt{l}'$ and this location contains the @Construct property, then the object pointed to by this location could be used as a constructor. The rule (E-NEW-CONSTR) deals with this case. In this rule, the @Construct auxiliary function is called to invoke the constructor.

The @Construct function (given in Figure 4.17) follows the following steps:

1. Given the constructor object location, $\mathtt{l}_1$, the rules first extract the *"prototype"* property of the object pointed to by the location $\mathtt{l}_1$. The returned value is stored in the location $\mathtt{ov}_2$.

2. If the returned prototype value is a location (as in (I-CONSOBJRET) and (I-CONSOBJNORM)), a new object is created with its @Proto property to be $\mathtt{ov}_2$ at location $\mathtt{l}_3$.

3. If the returned prototype value is not a location (as in (I-CONSGLOBRET) and (I-CONSGLOBNORM)), then the predefined location, ♯ObjectProt, which points to the object prototype is used as a prototype for the new object created at location $\mathtt{l}_3$.

4. The object created at location $\mathtt{l}_3$ is assigned to the this property when the constructor function is executed using the @Call function.

5. When the @Call function returns, the return object value, ov, is checked to see if it is a location or not. If it is a location, then the context returned as is (as seen in rules (I-CONSOBJRET) and (I-CONSGLOBRET)). If ov is not a location, then the location $\mathtt{l}_3$ is returned (as in rules (I-CONSOBJNORM) and (I-CONSGLOBNORM)).

$$\frac{\mathtt{l} : \mathtt{ov}' \; !< \; \mathtt{H} \quad \mathtt{H}_1 = \mathtt{H}, \mathtt{l} : \mathtt{ov}}{\mathrm{alloc}(\mathtt{H}, \mathtt{ov}) = \mathtt{H}_1, \mathtt{l}} \text{ (H-ALLOC)} \qquad \frac{\mathtt{l} : \mathtt{ov} \; < \; \mathtt{H}}{\mathtt{H}(\mathtt{l}) = \mathtt{ov}} \text{ (H-RET)}$$

$$\frac{\mathtt{i} \; !< \; \mathtt{i} \sim}{\mathtt{i} \; !< \; \{(\mathtt{i} : \mathtt{ov}) \sim\}} \text{ (H-NOTIN)} \qquad \frac{\mathtt{i} \; < \; \mathtt{i} \sim}{\mathtt{i} \; < \; \{(\mathtt{i} : \mathtt{ov}) \sim\}} \text{ (H-ISIN)}$$

$$\frac{\begin{array}{l}\mathtt{o} = \{(\mathtt{i}_1 : \mathtt{ov}_1) \sim [, \mathtt{i} : \mathtt{ov}_0], (\mathtt{i}_2 : \mathtt{ov}_2) \sim\} \\ \mathtt{o}_1 = \{(\mathtt{i}_1 : \mathtt{ov}_1) \sim [, \mathtt{i} : \mathtt{ov}], (\mathtt{i}_2 : \mathtt{ov}_2) \sim\} \\ \mathtt{H}(\mathtt{l}) = \mathtt{o} \qquad \mathtt{H}_1 = \mathtt{H}[\mathtt{l} \leftarrow \mathtt{o}_1] \qquad \mathtt{i} \; !< \; \mathtt{i}_1 \sim, \mathtt{i}_2 \sim\end{array}}{\mathtt{H}(\mathtt{l}.\mathtt{i} = \mathtt{ov}) = \mathtt{H}_1} \text{ (H-SET)}$$

$$\{(\mathtt{i}_1 : \mathtt{ov}_1) \sim, \mathtt{i} : \mathtt{ov}, (\mathtt{i}_2 : \mathtt{ov}_2) \sim\}.\mathtt{i} = \mathtt{ov} [\text{(H-GET)}]$$

Figure 4.18: Heaps and Objects

Auxiliary and Internal Property Semantics

The semantic rules in Figure 4.18 allow the semantic rules to access the abstract heap contents. If a location l is not in the heap, then (H-ALLOC) creates the new location and allocates the object value to this location. The rule (H-RET) returns the object value if the location already exists in the heap. The (H-NOTIN) and (H-IN) rules search an object for the occurrence of property names. The (H-SET) rule replaces an object value associated with a property. The (H-GET) rule get the object value associated with a property.

$$\frac{\mathtt{H}, \mathtt{l}.@\mathrm{HasProperty}(\mathtt{i})}{\mathrm{Scope}(\mathtt{H}, \mathtt{l}, \mathtt{i}) = \mathtt{l}} \text{ (SCOPE REF)}$$

$$\frac{\neg(\mathtt{H}.\mathtt{l}.@\mathrm{HasProperty}(\mathtt{i})) \quad \mathtt{H}(\mathtt{l}).@\mathtt{Scope} = \mathtt{l}'}{\mathrm{Scope}(\mathtt{H}, \mathtt{l}, \mathtt{i}) = \mathrm{Scope}(\mathtt{H}, \mathtt{l}', \mathtt{i})} \text{ (SCOPE LOOKUP)}$$

$$\mathrm{Scope}(\mathtt{H}, \mathrm{NULL}, \mathtt{i}) = \mathrm{NULL} (\text{SCOPE NULL})$$

$$\frac{\mathtt{i} \; < \; \mathtt{H}(\mathtt{l})}{\mathrm{Prototype}(\mathtt{H}, \mathtt{l}, \mathtt{i}) = \mathtt{l}} \text{ (PROTOTYPE REF)}$$

$$\frac{\mathtt{i} \; !< \; \mathtt{H}(\mathtt{l}) \quad \mathtt{H}(\mathtt{l}).@\mathtt{Proto} = \mathtt{l}'}{\mathrm{Prototype}(\mathtt{H}, \mathtt{l}, \mathtt{i}) = \mathrm{Prototype}(\mathtt{H}, \mathtt{l}', \mathtt{i})} \text{ (PROTOTYPE LOOKUP)}$$

$$\mathrm{Prototype}(\mathtt{H}, \mathrm{NULL}, \mathtt{i}) = \mathrm{NULL} \quad \text{(PROTOTYPE NULL)}$$

$$\mathtt{H}, \mathtt{l}.@\mathrm{HasProperty}(\mathtt{i}) = (\mathrm{Prototype}(\mathtt{H}, \mathtt{l}, \mathtt{i}) \neq \mathrm{NULL}) \quad \text{(I - HAS PROPERTY)}$$

Figure 4.19: Scope and Prototype Lookup

Figure 4.19 specifies the function for accessing the scope and prototype chains

for the presence of properties. In Figure 4.19, the relations $<$ and $!<$ check for presence and absence of a property value in the local object heap represented by the current location $\mathtt{l}$. These relations are defined as rules (H-NOTIN) and (H-ISIN) in Figure 4.18.

$$\frac{\text{Prototype}(\mathtt{H},\mathtt{l},\mathtt{m}) = \mathtt{l}_1 \quad \mathtt{H}(\mathtt{l}_1).\mathtt{m} = \mathtt{ov}}{\mathtt{H},\mathtt{l}.@\text{Get}(\mathtt{m}) = \mathtt{ov}} \text{ (I-GET)}$$

$$\frac{\text{Prototype}(\mathtt{H},\mathtt{l},\mathtt{m}) = \text{NULL}}{\mathtt{H},\mathtt{l}.@\text{Get}(\mathtt{m}) = \mathtt{undefined}} \text{ (I-GET-NULL)}$$

$$\frac{\text{Prototype}(\mathtt{H},\mathtt{l},\mathtt{m}) = \mathtt{l}_1}{\mathtt{H},\mathtt{l}.@\text{GetParent}(\mathtt{m}) = \mathtt{l}_1} \text{ (I-GETPARENT)}$$

$$\frac{\text{Prototype}(\mathtt{H},\mathtt{l},\mathtt{m}) = \text{NULL}}{\mathtt{H},\mathtt{l}.@\text{GetParent}(\mathtt{m}) = \text{NULL}} \text{ (I-GETPARENT-NULL)}$$

$$\frac{\text{Prototype}(\mathtt{H},\mathtt{l},\mathtt{m}) = \text{NULL}}{\mathtt{H},\mathtt{l}.@\text{Get}(\mathtt{m}) = \text{NULL}} \text{ (I-GET-NULL)}$$

$$\frac{\mathtt{m} \mathrel{!<} \mathtt{H}(\mathtt{l}_1) \quad \mathtt{H}(\mathtt{l}_1.\mathtt{m} = \mathtt{ov}) = \mathtt{H}_1}{\mathtt{H}, @\text{Put}(\mathtt{l}_1 * \mathtt{m}, \mathtt{ov}) \rightarrow \mathtt{H}_1, \mathtt{l}_1, \mathtt{ov}} \text{ (I-PUT)}$$

$$\frac{\mathtt{m} \mathrel{!<} \mathtt{H}(\mathtt{l}_1) \quad \mathtt{H}(\mathtt{l}_g.\mathtt{m} = \mathtt{ov}) = \mathtt{H}_1}{\mathtt{H}, @\text{Put}(\text{NULL} * \mathtt{m}, \mathtt{ov}) \rightarrow \mathtt{H}_1, \mathtt{l}_1, \mathtt{ov}} \text{ (I-PUT-GLOB)}$$

$$@\text{GetStrPrim}(\mathtt{H},\mathtt{l}) = \begin{cases} \mathtt{H},\mathtt{l}.@\text{Get}(\text{“@PrimitiveVal”}) & if\ \mathtt{H}.\text{IsStrPrim}(\mathtt{l}) \\ \mathtt{H},\mathtt{H}(\mathtt{l}).@\text{Get}(\text{“@PrimitiveVal”}) & if\ \text{Type}(\mathtt{H}(\mathtt{l})) \\ & \quad = \text{OBJECT} \\ & \wedge\ \mathtt{H}.\text{IsStrPrim}(\mathtt{H}(\mathtt{l})) \\ \text{“ ”} & otherwise \end{cases}$$

Figure 4.20: Internal Properties

Figure 4.20 defines the different internal properties used to access the object properties given an property name or object location, return the property's parent, or to insert a new property or a property value into the heap.

Figure 4.21 specifies functions for accessing the types of the different object values used in the abstract heap and the semantic rules. An object value could be a NULL type representing the `undefined` or `null`, an OBJECT type representing a location, a REFERENCE type representing a reference value, a TAINT type representing a security type, or a STRING type.

Figures 4.22 and 4.23 define the several taint manipulation operations required for the semantics. The @JoinTaint function (Figure 4.22) is a commutative func-

$\text{Type} : \texttt{ov} \rightarrow T$
$\text{Type}(\textsc{Null}) = \textsc{Null}$
$\text{Type}(\texttt{l}) = \textsc{Object}$
$\text{Type}(\texttt{ln} * \texttt{m}) = \textsc{Reference}$
$\text{Type}(\texttt{sectype}) = \textsc{Taint}$
$\text{Type}(\texttt{m}) = \textsc{String}$

$\text{IsPrim} : \texttt{ov} \rightarrow \texttt{bool}$
$\text{IsPrim}(\texttt{ov}) = \text{Type}(ov) \notin \{\textsc{Reference}, \textsc{Object}, \textsc{Taint}\}$

$\text{IsStrPrim} : \texttt{H}, \texttt{l} \rightarrow \texttt{bool}$
$$\text{IsStrPrim}(\texttt{H}, \texttt{l}) = \begin{cases} \&\texttt{false} & \texttt{l} == \textsc{Null} \\ \&\texttt{true} & (\texttt{H}, \texttt{l}.@\text{Get}(\text{“}@\texttt{PrimitiveVal}\text{”})! = \textsc{Null}) \\ \&\texttt{false} & otherwise \end{cases}$$
$\text{isActivation} : \texttt{H}, \texttt{l} \rightarrow \texttt{bool}$
$$\text{isActivation}(\texttt{H}, \texttt{l}) = \begin{cases} \&\texttt{false} & \texttt{l} == \textsc{Null} \\ \&\texttt{true} & (\texttt{H}, \texttt{l}.@\text{Get}(\text{“}@\texttt{IsActivation}\text{”})! = \textsc{Null}) \\ \&\texttt{false} & otherwise \end{cases}$$
$\text{IsFuncDecl} : \texttt{H}, \texttt{l} \rightarrow \texttt{bool}$
$$\text{IsFuncDecl}(\texttt{H}, \texttt{l}) = \begin{cases} \&\texttt{false} & \texttt{l} == \textsc{Null} \\ \&\texttt{true} & (\texttt{H}, \texttt{l}.@\text{Get}(\text{“}@\texttt{Body}\text{”})! = \textsc{Null}) \\ \&\texttt{false} & otherwise \end{cases}$$
$@\text{ToString} : \texttt{i}^{js} \rightarrow \texttt{m}$
$$@\text{ToString}(i^{js}) = \begin{cases} i^{js} & if\ \text{Type}(i^{js}) = \textsc{String} \\ \text{“}i^{js}\text{”} & otherwise \end{cases}$$

Figure 4.21: Types

argument 1	argument 2	result
NULL	sectype	sectype
⟨ NULL, LOW ⟩	⟨ NULL, LOW ⟩	⟨ NULL, LOW ⟩
⟨ NULL, LOW ⟩	⟨ m, HIGH ⟩	⟨ m, HIGH ⟩
⟨ $\texttt{m}_1$, HIGH ⟩	⟨ $\texttt{m}_2$, HIGH ⟩	⟨ {$\texttt{m}_1$,$\texttt{m}_2$}, HIGH ⟩

Figure 4.22: @JoinTaint function(commutative)

tion which is used to merge two security types. Notice that the LOW security type is always associated with a NULL string. The @GetTaint function is used to get the taint value of an object. The @AddTaints function is used to join the taint values of several objects and add it to the specified object.

$$
\frac{\begin{array}{c}
\text{@GetTaint}(\mathtt{H}, \mathtt{ov}) = \begin{cases} \mathtt{ov} & if\ \text{Type}(\mathtt{ov}) = \textsc{Taint} \\ \langle \textsc{Null}, \textsc{Low} \rangle & if\ \text{Type}(\mathtt{ov}) = \textsc{Null} \\ (\mathtt{H}, \mathtt{ov}.\text{@Get}(\text{“@Taint”})) & otherwise \end{cases} \\
\mathtt{sectype}_0 = \text{@GetTaint}(\mathtt{H}, \mathtt{l}) \\
\mathtt{sectype}_1 = \text{@GetTaint}(\mathtt{H}, \mathtt{ov}_1) \\
\vdots \\
\mathtt{sectype}_n = \text{@GetTaint}(\mathtt{H}, \mathtt{ov}_n) \\
\mathtt{sectype}'_1 = \text{@JoinTaint}(\mathtt{sectype}_0, \mathtt{sectype}_1) \\
\vdots \\
\mathtt{sectype}'_n = \text{@JoinTaint}(\mathtt{sectype}'_{(n-1)}, \mathtt{sectype}_n) \\
\mathtt{H}, \mathtt{l}.\text{@Put}(\text{“@Taint”}, \mathtt{sectype}'_n) \rightarrow \mathtt{H}_1, \mathtt{l}, \mathtt{sectype}'_n
\end{array}}{\mathtt{H}, \mathtt{l}.\text{@AddTaints}([\mathtt{ov}_1, \ldots, \mathtt{ov}_n]) \rightarrow \mathtt{H}_1, \mathtt{l}} \text{ (H-ADDTAINTS)}
$$

Figure 4.23: Taint Manipulation Operations

Figures 4.24 and 4.25 specify several pre-defined templates used to create different kinds of objects by the operational semantic rules.

$$
\begin{array}{l}
\text{new\_object}(\mathtt{m}, \mathtt{l}, \mathtt{sectype}) = \{ \\
\qquad \text{“@Class”} : \mathtt{m}, \\
\qquad \text{“@Proto”} : \mathtt{l}, \\
\qquad \text{“@Taint”} : \mathtt{sectype} \\
\qquad \} \\
\\
\text{new\_proto}(\mathtt{m}, \mathtt{l}_1, \mathtt{l}_2, \mathtt{sectype}) = \{ \\
\qquad \text{“@Class”} : \mathtt{m}, \\
\qquad \text{“@Proto”} : \mathtt{l}_1, \\
\qquad \text{“}constructor\text{”} : \mathtt{l}_2 \\
\qquad \text{“@Taint”} : \mathtt{sectype} \\
\qquad \} \\
\\
\text{new\_primitive}(\mathtt{m}, \mathtt{l}, \mathtt{sectype}) = \{ \\
\qquad \text{“@Class”} : \mathtt{m}, \\
\qquad \text{“@Proto”} : \mathtt{l}, \\
\qquad \text{“@Taint”} : \mathtt{sectype} \\
\qquad \}
\end{array}
$$

Figure 4.24: Predefined Object Templates-1

new_string_primitive($\mathtt{m}, \mathtt{l}, \mathtt{m}_1, \mathtt{sectype}$) = {
“@Class” : m,
“@Proto” : l,
“@Taint” : sectype,
“@PrimitiveVal” : $\mathtt{m}_1$ }

new_function(“fun([id ∼]){*prog*}”, $\mathtt{l}_1, \mathtt{l}_2, \mathtt{sectype}$) = {
“prototype” : $\mathtt{l}_2$,
“@Proto” : FunctionProt,
“@Class” : “*Function*”,
“@Call” : true,
“@Construct” : true,
“@FScope” : $\mathtt{l}_1$,
“@Body” : “fun([id ∼]){*prog*}”,
“*length*” :| id ∼|
“@Taint” : sectype
}

new_arguments(n, ([$\mathtt{ov}_1, \ldots, \mathtt{ov}_n$]), l, sectype) = {
[“1” : $\mathtt{ov}_1$,
...
“n” : $\mathtt{ov}_n$,]
“@argumentsFlag” : true,
“@Proto” : ObjectProt,
“*callee*” : l,
“*length*” : n
“@Taint” : sectype
}

new_activation($\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3$, sectype) = {
“@Scope” : $\mathtt{l}_3$,
“@Proto” : NULL,
“@IsActivation” : true,
“@this” : $\mathtt{l}_2$,
“*arguments*” : $\mathtt{l}_1$,
“@Taint” : sectype
}

Figure 4.25: Predefined Object Templates-2

4.3.5 Handling the Features of JavaScript

Figure 4.26 gives an example of a sample JavaScript heap computed using the VEX analysis. Every object and function in the JavaScript program is represented as a node in the heap, while the properties of the object are represented using edges in the graph. In the figure, the global object `loc_Global` has five properties `ObjectProt`, `FunctionProt`, `Array`, `ArrayProt`, and `array_instance` pointing to the nodes `loc_ObjProt`, `loc_FunProt`, `loc_1`, `loc_ArrayProt`, and `loc_4` respectively. Every node in the heap is associated with a taint value, HIGH or LOW – HIGH representing the untrusted objects and LOW representing the trusted objects. High taints and low taints are represented by red and blue nodes, respectively, in the figures (all nodes in Figure 4.26 are LOW). Figure 4.27 shows the *initial* abstract heap representation of the `window.content.document` object and the `window.document` object; notice that one of the nodes `loc_document`, has a high taint.

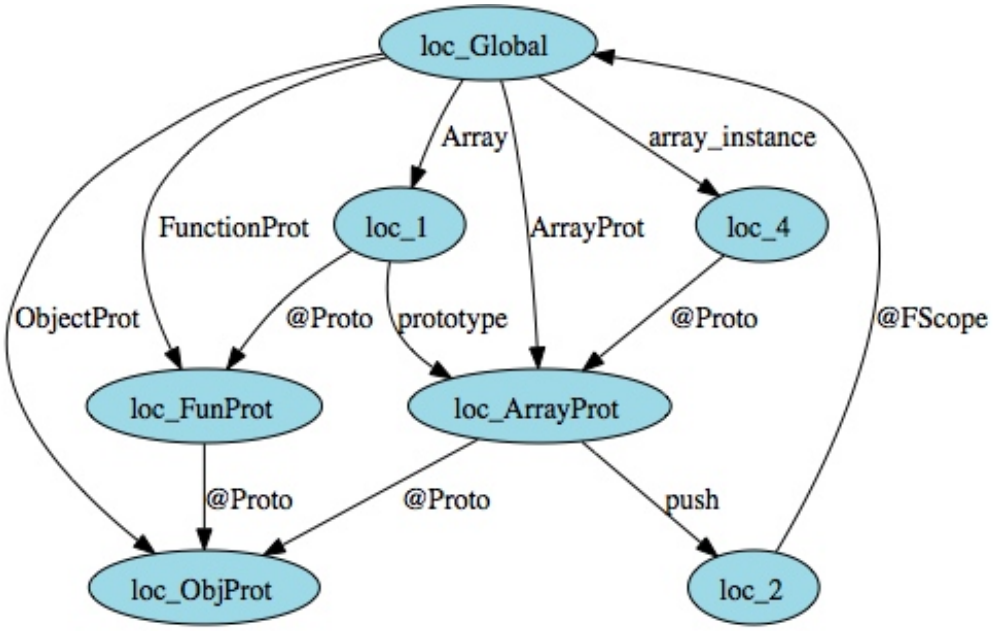

Figure 4.26: Sample JavaScript Heap – Array Object

Prototype-based Inheritance: Figure 4.26 illustrates how VEX handles prototype-based inheritance. The `Array` object in JavaScript is represented as the node `loc_1` in the figure. Since the `Array` object is a constructor, which can be used to create new instances of the array, it has a `prototype` field pointing to the object, `ArrayProt`, represented in the graph by the node `loc_ArrayProt`. A

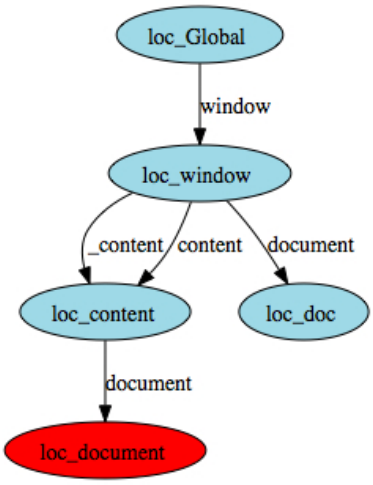

Figure 4.27: window.content.document Object

new `Array` instance, `array_instance` object, is created in the program using the statement: `array_instance = new Array ()`. In Figure 4.26, `loc_4` represents the `array_instance` object. The `@Proto` field of this object points to the object `loc_ArrayProt`. Therefore, the `push` method is accessible to the `array_instance` object and can be called using the `array_instance.push`.

Function and Object Summaries: Natively supported functions and objects are replaced with stubs that summarize the effect on the heap and the taints when accessing them. VEX function and object summaries are hence simplified JavaScript objects and functions containing only the essential functionality of the objects. For example, a JavaScript `Array` object is defined in Figure 4.28 to be a function object with the `@Class`, `prototype`, and `@Proto` properties initialized to the string "Function", identifier `ArrayProt`, and identifier `FunctionProt`, respectively. The variables `FunctionProt` and `ArrayProt` point to the prototype objects, which contain the various functions like `length` and `push`.

```
var Array = function(){
  this.@Class = "function";
  this.prototype = ArrayProt;
  this.@Proto = FunctionProt; };
```

Figure 4.28: `Array` object summary in VEX

Browser's DOM API and XPCOM components: VEX treats most of the browser's DOM API and XPCOM components as uninitialized variables, fields, and

functions. However, VEX provides explicit function summaries for the API components and objects that VEX needs to keep track of in order to trace the flows to and from the objects. VEX analysis sets the taint of the objects that represent insecure sources or those that are dependent on insecure sources to HIGH.

Higher-order functions: VEX analysis accurately keeps track of the objects and implements function calls by inlining the function bodies according to the JavaScript semantics. Higher-order functions calls are also inlined. Additionally, VEX provides summaries for some higher order functions in the JavaScript API. For example, the `settimeout` function in JavaScript takes a callback function as its first argument. This function is represented in VEX as a function in which the function body invokes the callback function in the first argument.

Dynamically generated code: The `eval` method in JavaScript allows execution of dynamically formed code, and is widely used in browser extensions. While an accurate analysis of the structure of dynamically created code is a research topic in itself, and out of the scope of this paper, the analysis cannot simply ignore `eval` statements. VEX analysis performs a *constant-string analysis* for strings and string operations. If the actual parameters to the `eval` statement evaluate to a constant string, VEX's static analysis engine parses these constant strings and inserts them into the program flow just after the `eval` statement. This ensures that these newly parsed statements are included in the computation of the taint. In most correct extensions, an `eval`-ed statement is dynamically chosen from a set of constant-strings or taken from trusted sources, and hence evaluate to a constant string on the path explored (and tracked accurately by VEX). Parameters to `eval`, whose exact string values are not statically inferred by VEX along the path explored, are tested to check if they are tainted. If there is a flow from an *untrusted* source to an `eval`, VEX will report this flow, as it corresponds to a vulnerable flow pattern.

Object properties accessed in the form of associative arrays: In JavaScript, objects are treated as associative arrays. This means that any property of the object can be accessed using the array notation. Array indices could be constant strings, which are then evaluated to get the actual property being accessed; or they could be numbers, which indicate the property number that is being accessed; or they could be variables, that could be instantiated at run time. If VEX cannot evaluate the array index to a property name for any reason, the array access conservatively gets the taints of every property in the parent array object.

Functions that take arbitrary number of arguments: Some functions in JavaScript can have variable numbers of arguments. For example, the `push` method of an array can be called with any number of arguments and the arguments will be appended to the end of that array. To handle this in VEX, the object representing the `push` method has a special property indicating that it can take a variable number of arguments and when the method is called, VEX analysis conservatively appends the taints of all the arguments to the `push` method to the array object on which the method is called.

Initialization of the Analysis: The functions in the Firefox extensions are usually event handlers triggered by the user events like mouse clicks on menu items, keyboard events, etc. The calls to the event handler are embedded in a special file written in XUL, the special markup language used to create the user interface of an extension. As a pre-processing step of our analysis, we extract all the event calls from the XUL files and write them into a single JavaScript file along with all the JavaScript files in the extension. This ensures that we trigger all the events in the extension. However, this has a side effect of considering only a single program execution sequence.

4.3.6 A Note on Soundness

Most static analysis tools, such as those used in compilers and those used in abstract interpretation, *over-approximate* the concrete semantics, and hence are *sound*. In the context of flow analysis, a sound tool never reports that a program has no flows when it has one. Soundness often entails a large number of *false positives*, i.e. flows that the tool reports but may not actually ever happen during execution.

VEX is not sound. We believe that a sound state-of-the-art analysis tool for JavaScript extensions would overwhelm and frustrate the tool's users with a torrent of false positives. Thus to handle certain features of JavaScript without producing a lot of false positives, we chose not to make VEX sound. As a consequence, for example, a maliciously written extension could easily evade detection by VEX. On the other hand, a maliciously written extension can easily harm its users directly, without any input from untrusted web pages. This underlies the reason why our threat model assumes that the extension author is not malicious.

Instead of aiming for soundness, we concentrated on making VEX accurate on

paths in the program, without *collapsing* (merging) the nodes of the heap in any way. Since VEX can only analyze a finite number of paths in the program (obtained by unrolling recursion a bounded number of times) in this accurate manner, the analysis VEX performs is inherently not sound.

False positives are also, of course, still possible in VEX, i.e. VEX may report flows that actually do not exist in the program. This stems from the fact that the analysis uses an abstraction. In particular, not having precise enough information for evaluating conditionals, not precisely being able to determine the values of strings being subject to `eval` statements, etc. are common sources for false positives. Compared to classical heap analysis in programs that merges nodes in heaps, VEX performs a much more accurate analysis that reduces the number of false positives considerably. In experiments, we found that VEX produces very few false positives.

Overall, our choices were determined mainly by the complexity of JavaScript analysis and our aim at building a *useful* tool, which in turn led us to sacrifice soundness.

4.4 Implementation and Evaluation

VEX is implemented in Java ($\sim$ 7000 LOC), and utilizes a JavaScript parser built using the ANTLR parser generator for the JavaScript 1.5 grammar provided by ANTLR [65]. ANTLR outputs Java-based Abstract Syntax Trees (AST) for JavaScript sources obtained from the pre-processing of the extension's XUL and JavaScript files. The XUL files add different UI elements to the browser's chrome. When any one of the user-interface elements is invoked and clicked, the corresponding event is triggered and the event-handler is called. We extract all such calls to the event-handlers from the XUL files and run them using VEX's abstract operational semantics.

During the execution of the program using the abstract operational semantics outlined in Section 4.3, if the program reaches a vulnerable sink, it checks if the inputs or assignments to the sink are tainted. If they are tainted, VEX reports the occurrence of the flow along with the source objects and sink locations in the code. The source objects are the objects described in Section 4.2 and the sink locations are the points where the sinks described in Section 4.2 are encountered during the execution. The rest of this section summarizes our results.

The number of loop unrollings can be set as a parameter in the VEX analysis engine (in our experiments, a bound of just *one* was used). The VEX implementation has a number of optimizations to improve memory usage and speed. To save memory, abstract heaps are freed when backtracking in the depth-first search. But to save time, abstract heaps at join points are cached and compared when other paths hit these points, to avoid exploring paths unnecessarily.

4.4.1 Evaluation Methodology

The extensions we analyzed were chosen as follows. First, in October 2008, we built a suite of extensions using a random sample of 1828 extensions from the Mozilla add-ons web site, by downloading the first extensions in alphabetical order for all subject categories. This extension suite had 2 extensions with known vulnerabilities. In November 2009, we downloaded 699 of the most popular extensions and 8 extensions with known vulnerabilities. In April 2011, we downloaded 2082 popular extensions. The random sample and the popular extensions had 314 extensions in common, for a total of **4303 extensions**. Our suite includes *multiple versions* of some extensions, allowing cross-version comparisons. For instance, we found a new version of the FIZZLE, to be vulnerable even though its authors tried to fix the vulnerabilities in the previous version.

We extracted the JavaScript files from these extensions and ran VEX on them, using a 2.4GHz 64 bit x86 processor with a maximum heap size of 16GB for the JVM.

To evaluate the effectiveness of VEX, we perform two kinds of experiments. First, we run VEX on the downloaded extensions and check if any of them have one of the malicious flow patterns. Second, we check if VEX can detect known extension vulnerabilities.

4.4.2 Experimental Results

Finding flows from injectible sources to executable sinks: Figure 4.29 summarizes the experimental results for flows that are from injectible sources to executable sinks (flows for which the sinks are `eval` and `innerHTML`). Of the 4303 extensions analyzed by VEX, a `grep` showed that a total of 1498 extensions had the occurrence of either the string "`eval` (" or the string "`innerHTML`" or both.

Flow Pattern	VEX Alerts	Attackable			Not Attackable		
		Confirmed	*Source from User/Extension /System*	*Source is a trusted website*	*Sanitized input*	*Non-chrome sinks*	*Non-existent flows*
Content Doc to `eval`	17	2*	12	0	0	0	3
Prefs to `eval`	12	1*	9	2	0	0	0
Unknown var to `eval`	36	0	16	2	3	0	15
Content Doc to `innerHTML`	25	1*	0	4	3	15	2
RDF to `innerHTML`	7	5*	1	1	0	0	0
Prefs to `innerHTML`	12	0	11	1	0	0	0
`popupNode` to `innerHTML`	2	0	0	0	1	1	0
Unknown to `innerHTML`	42	0	26	7	3	5	1
Total	153	9	75	17	10	21	21

* Attackable Extensions are listed in Section 4.4.2

Figure 4.29: Flows from Injectible Sources to Executable Sinks.

The first column of Figure 4.29 indicates the exact source to sink flow pattern checked by VEX. The second column indicates the number of extensions on which VEX reports an alert with corresponding flows. On an average, VEX took 11.5 seconds per extension. It took about a week to analyze all the extensions with flows from untrusted sources to `eval` and `innerHTML` sinks.

To look for potential attacks, we manually analyzed the extensions with suspect flows found by VEX, spending about 20 minutes per extension on average. The next column reports the number of extensions on which we could engineer an attack based on the flows reported by VEX. We were able to attack nine extensions, of which only two extensions (FIZZLE VERSION 0.5 and BEATNIK V-1.0) were already known to be vulnerable. The rest of the attacks are new.

The next column shows the extensions where the source is provided either by the extension user or the extension developer or computed from the system parameters by the extension. The values are either stored in the preferences or in a local file. Since we trust the users and extension developers in our trust model, these extensions are considered to be non-vulnerable. However, if the preferences file or the local file system is corrupted in any way, these extensions can be attacked.

The fifth column shows the extensions where the source is code from a web site, and where an attack *is possible* provided the web site can be attacked. In other words, these extensions rely on a *trusted web site* assumption (e.g., that the code on the Facebook website is safe). We think that these are valid warnings that users of an extension (and Mozilla) should be aware of; trusted web sites can after all be compromised, and the code on these sites can be changed leading to an attack on all users of such an extension.

Not all flows lead to attacks— the next set of columns describe the alerts that we were unable to convert to concrete attacks. Some extensions were not exploitable as the input is *sanitized* correctly (either by the extension or the browser), preventing JavaScript injection. Others extensions were not exploitable as the sinks were not in chrome executable contexts. These extensions are noted in the next two columns. Finally, VEX, being a static flow-analysis tool, does report alerts about flows that do not actually exist— there were very few of these, and are noted under the column "Non-existent flows". Section 4.4.4 discusses the flows that do not lead to attacks.

New vulnerabilities discovered: The number of security vulnerabilities discovered are shown in column 3 in Figure 4.29, of which 7 are new. WIKIPEDIA TOOL-

BAR versions v-0.5.7 and v-0.5.9 have flows from `window.content.document` to `eval`, which leads to attacks. MOUSE GESTURES REDOX v-2.0.3 has flows from `nsIPrefService` to `eval`, which also led to an attack. BEATNIK v-1.2, FIZZLE v-0.5.1, and FIZZLE v-0.5.2 are also attackable, and have flows from `nsIRDFService` to `innerHTML`. KAIZOU v-0.5.8 has a flow from `window.content.document` to `innerHTML` which leads to attacks. Section 4.4.3 gives some details about the the flows and the attacks in some of the vulnerable extensions.

Known vulnerabilities detected: Apart from the new vulnerabilities found by VEX, there are several extensions that have been reported to be vulnerable in the past. In the course of our research, we found 18 unique extensions that were reported to be vulnerable in various databases like CVE, Secunia, etc. Of these 18, we did not find the source code for five extensions (GREASEMONKEY v$\leq$0.3.5, WIZZ RSS v$<$3.1.0.0, SKYPE v$\leq$3.8.0.188, MOUSEOVERDICTIONARY v$<$0.6.2, POW v$<$0.0.9), so we did not analyze them. Of the remaining 13 extensions, we found that 10 of them can potentially be found using explicit information flow analysis techniques, like VEX.

Currently, VEX can detect 5 of the above 10 known extension that have flow-based vulnerabilities: FIZZLE v-0.5, BEATNIK v-1.0, COOLPREVIEWS v-2.7, 2.7.2, INFORSS v-$<$=1.1.4.2, and SAGE v- $<$ 1.3.9, $<$=1.4.3. COOLPREVIEWS has flows from `document.popupNode` to `appendChild`. INFORSS has flows from `nsIRDFService` to `appendChild`. SAGE has flows from `BookmarksUtils` to an object accessing the local file system using the `nsIFile` interface.

The remaining 5 extensions have flow vulnerabilities but were not found by VEX for the following reasons. For FEEDSIDEBAR v$<$ 3.2, FIREBUG v-1.01, SCRIBEFIRE v$<$= 3.4.2, and UPDATE SCANNER v¡ 3.0.3 the trigger of the flow was in an event handler or a function call which was called outside the extension's code base. In YOONO version $\leq$ 6.1.1 an un-sanitized JavaScript element like an image or link is rendered in the chrome context. However, it was difficult to find the source and sink objects from its source code.

Finally, there were three extension vulnerabilities (for which we had the source) that cannot be found by VEX because they are not flow vulnerabilities. These vulnerabilities include attacks on a file server (e.g.,FIREFTP v$<$0.97.2, $<$ 1.04), and directory traversal attacks (e.g.,NAVIGATIONAL SOUNDS version-1.0.2, AJAX YAHOO MAIL VIAMATIC WEBMAIL version-0.9) when a chrome package is "flat" rather than contained in a .jar. In both the above cases, an attacker can escape from the

extension's directory and read files in a predictable location on the disk. Since such attacks are not related to chrome privilege escalations, and VEX does not handle them.

4.4.3 Successful Attacks

Attack scripts: All our attack scenarios involve a user who has installed a vulnerable extension who visits a malicious page, and either automatically or through invoking the extension, triggers script written on the malicious page to execute in the chrome context. Figure 4.30 illustrates an attack payload that can be used in such attacks: this script displays the folders and files in the root directory. The attack payloads could be much more dangerous, where the attacker could gain complete control of the affected computer using XPCOM API functions. More examples of such payloads are enumerated in the white-paper given in [66]. In this section, we illustrate a few attacks on extensions with previously unknown vulnerabilities.

```
<script>
var root = Components.classes
["@mozilla.org/file/local;1"].createInstance
(Components.interfaces.nsILocalFile);
try {
root.initWithPath("/."); // for Linux or Mac
}catch (er){
root.initWithPath("\\\\."); // for Windows
}
var drivesEnum = root.directoryEntries, drives = [];
while (drivesEnum.hasMoreElements()) {
drives.push(drivesEnum.getNext().
QueryInterface(Components.
interfaces.nsILocalFile).path);
}
alert (drives);
</script>
```

Figure 4.30: Attack Script to Display Directories

Wikipedia Toolbar, up to version 0.5.9: If a user visits a web page with the directory display attack script in its <head> tag, and clicks on one of the Wikipedia toolbar buttons (unwatch, purge, etc.), the script executes in the chrome context.

The attack works because the extension has the code given in Figure 4.31 in its toolbar.js file.

```
script = window._content.document.
getElementsByTagName(``script")[0].innerHTML;
eval (script);
```

Figure 4.31: Wikipedia Toolbar Code

The first line gets the first <script> element from the web page and executes it using `eval`. The extension developer assumes the user only clicks the buttons when a Wikipedia page is open, in which case <script> may not be malicious. But the user might be fooled by a malicious Wikipedia spoof page, or accidentally press the button on some other page. VEX led us to this previously unknown attack, which we reported to the developers, who acknowledged it, patched it, and released a new version. This resulted in a new **CVE vulnerability (CVE-2009-41-27).** The fix involved inserting a conditional in the program to check if the URL of the page is in Wikipedia's domain and evaluating the script only if this is true.

Fizzle versions 0.5, 0.5.1, 0.5.2

FIZZLE is a RSS/Atom feed reader that uses Livemark bookmark feeds. Vulnerability report CVE-2007-1678 explains that FIZZLE VER.0.5 allows remote attackers to inject arbitrary web scripts or HTML via RSS feeds. FIZZLE's RSS feeds are obtained from the bookmarks' RDF resource, using the XPCOM RDF service. The author of FIZZLE purportedly *fixed* this vulnerability in the next version; however, VEX signaled the presence of a flow, and we found that the sanitization routine that the programmer wrote was flawed, and the extension can be attacked using suitably encoded scripts. These new attacks for FIZZLE VER 0.5.1 and FIZZLE VER 0.5.2 were not known before, to the best of our knowledge.

Figure 4.32 gives a highly simplified version of FIZZLE, to show its information flows. When the user clicks on the FIZZLE extension toolbar to see the feeds, FIZZLE is initialized, i.e., `sys.startup()` on line 15 is called. This method loads the bookmarks from the Firefox bookmarks folder. The title and URL of the feeds are obtained from the bookmarks' RDF resource and then stored in an array in FIZZLE when `bookmarks.load()` is called. After the bookmarks are loaded, `ui.buildFeedList()` is called. In this method, the bookmark array is accessed on line 24 and the elements are added to a variable named `html`

bookmarks.js:

```
1. function Bookmarks(){
2.  var  bookmarks = new  Array();
3.  this.load = function(){
4.   bookmarks = new  Array();
5.   var  rdf = Components.classes[
       "@mozilla.org/rdf/rdf-service;1"]
       .getService(Components.interfaces.nsIRDFService);
6.   var  bmds = rdf.GetDataSource("rdf:bookmarks");
7.   var  iter = bmds.GetAllResources();
8.   while (iter.hasMoreElements()){
9.    var  element = iter.getNext();
10.  bookmarks.push(
       {name:element.name, url:element.url});
11. } } }
```

sys.js:

```
12.    var  sys = new  Sys();
13.    function Sys() {
14.     var  bookmarks = null;
15.     this.startup = function() {
16.      bookmarks = new  Bookmarks();
17.      bookmarks.load();
18.      ui.buildFeedList(); }
19.     this.getBookmarks(){
20.      return  bookmarks; } }
```

ui.js:

```
21.    var  ui = new  Ui();
22.    function Ui() {
23.     this.buildFeedList = function() {
24.      var  bm = sys.getBookmarks();
25.      for (var  i=0; i<bm.size(); i++) {
26.       var  mark = bm.get(i);
27.       html += <p> mark.name; }
28.      div.innerHTML = html; } }
```

Figure 4.32: FIZZLE vulnerability code.

on line 27. This `html` variable is then assigned to the `innerHTML` property of the $\langle div \rangle$ tag of an HTML page. This page is then displayed in a frame in the browser. The attack happens when a malicious RDF file is loaded, where the `name` element of the feed contains JavaScript. Assigning a specially crafted script to the `innerHTML` property at line 28 results in the script being executed under chrome privileges.

To detect this kind of attack, we must be able to determine that the information that flows into the `html` variable and eventually into the `innerHTML` property is from the bookmarks' RDF resource. It is difficult to detect this manually, because most extensions are encoded in many separate JavaScript files spread across multiple directories, and the routines defined in these files have complex interactions with each other. Even the example shown in Figure 4.32 is spread over three different JavaScript files, and we have omitted many lines of code from the functions shown. As mentioned earlier, VEX users can define summaries for library functions, or just rely on default summaries. Given a function summary for the `push` method of the `Array` object defined in the XPCOM library, VEX detects that FIZZLE has flows from the RDF service to `innerHTML`.

Beatnik version 1.2

BEATNIK is another RSS reader with the same kind of problematic flow as FIZZLE, documented in CVE-2007-3110 for BEATNIK version 1.0. In the Mozilla add-ons page for the subsequent version of BEATNIK, the extension developer said he had sanitized the RSS feed input. VEX found that there were still flows from the bookmarks' RDF to the `innerHTML` property in BEATNIK version 1.2, because VEX currently does not consider declassification via sanitization. Our manual examination showed the new sanitization to be inadequate. The sanitization parses the feed input and checks whether the nodes contain script. If the feed contains only text nodes, it is appended to the RSS feed title; otherwise it is discarded. By encoding the ⟨ and ⟩ tags as their HTML entity names, we can fool this routine. If we name the RSS feed as follows:

```
Title &lt; /a &gt&lt; img  src = &quot;&quot;
onerror= 'CODE FROM FIGURE 4.30' & gt; Beatnik
&lt;/img&gt; &lt; a &gt;
```

the string is converted into

```
Title </a> < img src =" " onerror= 'CODE FROM
FIGURE 4.30'> Beatnik </img> <a>
```

and results in an attack. To the best of our knowledge, this attack has not been reported thus far. One must understand the extension code to form these attack strings; in this case, the <a> tag had to be closed at the beginning of the string and opened again at the end for the script to work.

Kaizou v- 0.5.8: Kaizou is a web development extension that allows users to open the source of any web page in a separate window, modify the contents and render it again in the current window by pressing a button. However, this separate window has chrome privileges, and when the user saves the changes he made to the page source, the scripts in the page are executed with chrome privileges. A malicious web page can have an attack script, which could result in an attack when modified using KAIZOU.

Mouse Gestures Redox v-2.0.3: The MOUSE GESTURES REDOX extension allows users to create shortcuts for frequently used commands without using keyboard, menu or toolbars. The users can either create new gestures or download them from an online source. The new gestures are scripts, which are stored in the browser's preferences file. When the gestures are enabled, they are retrieved from the prefs.js file and sent as arguments to the `eval` () function, thereby activating the gestures. If any of the gestures downloaded from the internet contain attack scripts, they would be executed in the chrome context when `eval` is called.

4.4.4 Flows That Do Not Result in Attacks

Figure 4.33 gives several examples of the suspect flows that we manually analyzed and for which either trusted sources were assumed by the extension or we could not find attacks.

The first set has extensions accessing values from web-sites or sources it trusts, and the values flow to `eval` or `innerHTML`. Of course, if the trusted sources are compromised, then the extensions may become vulnerable. The second set illustrates examples where the input was sanitized between the source and the sink. We do not know for sure that the sanitization is adequate, but we were unable to attack it. The third set of extensions had non-chrome sinks. The last set has two examples that show false positives where the flows reported by VEX do not exist in the code.

Classification	Extension	Flow pattern	Explanation
Source is trusted web site	WORLD SMS V-2.2	Unknown var to `eval`	The source is a web site: `http://worldsms.co.cc/json?get=info`, which is hardcoded into the extension code.
	BROWSER BACKGROUNDS V-0.3.5	`nsIRDFService` to `innerHTML`	The user installs background images from the web site `http://browserbackgrounds.com/`.
Sanitized input	ALPHANUMERATOR V-0.2	Content Doc to `innerHTML`	The input string is converted to numbers effectively sanitizing it.
	VIEW SOURCE CHART V-2.7	Content Doc to `innerHTML`	Input HTML tags are sanitized into custom tags
Non-chrome sinks	PAGEDIFF V-1.3.0	Content Doc to `innerHTML`	The display page has a chrome url but is marked to be "content-type."
Non-existent flows	LINK_ALERT V-0.8.2.1	Unknown var to `eval`	`eval`'s argument is a packed and minified JavaScript string that VEX could not parse properly. VEX finds an unknown variable in this incorrectly parsed string.
	SKIPSCREEN V-0.1.09102009	Unknown var to `innerHTML`	During the execution, the extension will never follow the branch that has the sink, as the conditional variable is never initialized in the program.

Figure 4.33: Extensions That Could Not be Attacked

4.5 Related Work

4.5.1 Firefox Browser Extension Security

Several researchers developed proof-of-concept malicious extensions [67, 68] and demonstrated how they can be used to perform malicious operations in the browser. Verdurmen [67] described the extension architecuture and the extension review process. He created two proof of concept malicious extensions: a password stealer and an extension that performs unauthorized bank transactions.

Ter Louw *et al.* [68] highlight how a malicious extension can tamper with the Firefox browser's code and date. First, a malicious extension can damage the browser's codebase integrity by subverting the browsers installation process, taking control of the borwser, and hiding its presence completely. Second, it can also read and write users' confidential data, even if it sent over an encrypted connection. The authors developed a proof of concept malicious extension, BrowserSpy, to demonstrate these security risks posed by browser extensions. They propose a two-pronged solution for preventing these security risks. First, to prevent the extension from subverting the browser's code base, they propose to allow the extension users to sign code that they want to install in their browser. Additionally, they provide automated tools to allow the user to check the integrity of the code whenever the user wishes to do so. This ensures that any malicious extension cannot introduce new code into the signed extension's code base. Additionally, drive-by-downloads of extensions are not possible i.e. the extension cannot be modified without the user noticing the change. Second, they develop a run-time monitor for the extension code to ensure that extension does not tamper with or steal privileged resources and content. The authors specified 6 different policies providing various functionality like: allowing or denying access to a single XPCOM API, enforcing the same origin policy within the browser extension, denying access to the password manager, preventing URL history leaks, *etc*. Arbitrarily assigning of policies to extensions could lead to problems because even benign extensions which require the particular behavior can also be blocked. Therefore, the authors suggest that policies must be assigned on a per-extension basis. Assigning policies on a per-extension basis could become cumbersome; the user might not even have the knowledge of what kind of policy to use for a particular extension. Additionally, restricted interfaces can still be susceptible to security vulnerabilities if the policies are not assigned properly.

SABRE [21] is a framework for dynamically tracking in-browser information flows for analyzing JavaScript-based browser extensions. SABRE does this by associating a security label with every JavaScript object created in the browser memory. The paper provides a list of sources and sinks tested to check for confidentiality violations. They also have a separate set of sources and sinks to check for integrity violations. The taints are tracked by modifying the JavaScript interpreter such that it tracks information flows during the execution of each JavaScript instruction. The authors also modified other browser subsystems, including the DOM subsystem (e.g., HTML, XUL) and XPCOM, to store and propagate security labels to allow information flow tracking across browser subsystems. Finally, the JavaScript interpreter is equipped with a special declassification and endorsement mechanism. Whenever, SABRE reports an information flow violation, an vetter or a user must decide whether it is an actual security violation or an allowed extension behavior. In the former case, the user can whitelist the flow using the declassification mechanism. One main feature of SABRE's information flow tracking is it facilitates both explicit and certain forms of implicit flow analysis. Implicit flow analysis is required when the analysis is aiming to find a malicious extension.

Djeric *et al* [22] propose another approach to dynamically track taints in both the browser's native code and the script interpreter. Their taint tracking algorithm is extremely similar to that of SABRE, except that the taint checking decisions are used to allow or deny the execution of JavaScript operations. Therefore this techniques prevents vulnerabilities from being executed. Another notable difference between this approach and SABRE is that this approach tracks implicit flows only in untrusted content. The analysis tracks only the explicit flows in the trusted browser code and extension code. This ensures that the analysis does not generate a lot of false positives. However, this means that it only works on trusted code and will not be able to track malicious extensions that use control flows.

Although dynamic techniques are useful in preventing certain types of script injection attacks if they are enforced by the web browser, they suffer from a few drawbacks. First, they impose a performance and memory overhead on the browser because of the need to keep track of the security label for every JavaScript object inside the browser. When a questionable flow is detected dynamically, the browser has to either choose an appropriate action (which might be overly restrictive) or ask the user to choose an action (which might lead to an attack if the user chooses a wrong option).

Additionally, it is not possible to completely test all the behaviors of the browser extensions before deployment using dynamic techniques. This is due to the fact that it is not easy to create test inputs for JavaScript extension since each extension takes different kind of input and usually requires keyboard or mouse events to trigger the extension. Therefore, it is not feasible to find vulnerabilities before deployment by running the extension on various inputs. These drawbacks motivated us to find a static alternative to analyze browser extensions.

4.5.2 Security of Extensions to Other Browsers

One of the reasons for extension vulnerabilities in the Firefox web browser is that the browser architecture is too permissive – it allows every extension to have the same privileges as the browser. Barth *et al* [69] examined the 25 most popular Firefox extensions and found that 88% of them do not need the full set of privileges available to them. Additionally, 76% of the extensions used an unnecessarily powerful API, which made it difficult to automatically reduce their privileges. Based on these experiences, they proposed a new browser extension architecture which had and API for separating access to the different components based on the principles of least privilege and privilege separation. The Google Chrome extension system adopted this new architecture.

Having a good extension architecture is not enough, however. Guha *et al* [70] conducted an analysis of the Chrome extension manifest files and found that this extension model is not entirely effective in limiting the privileges given to the browser extensions. This is partly because the API still provide more privileges than required by the extension. They studied the page access control behavior of 1,139 extensions. They found that only 17% of the extensions correctly limit access to only the web pages that they need to access. The rest of the extensions either use overly permissive wildcard characters to limit page access or request permissions to access all web pages. This discussion strongly suggests that apart from the extension architecture and the APIs provided to the extensions, the access control policy and its implementation in the extension program play a major role in the security of the system. Guha *et al* [70] propose a new framework for authoring, analyzing, verifying, and deploying secure browser extensions.

4.5.3 Operational Semantics of JavaScript

Maffeis *et al* [39] proposed a small-step operational semantics for JavaScript, using which they analyze security properties of web applications. They also use their operational semantics for generating safe subsets of JavaScript and to manually prove that the so-called safe subsets of JavaScript are in fact vulnerable to certain attacks [51]. Our operational semantics follows their operational semantics, but works on an abstract heap. Guha *et al.* [38] propose an alternate operational semantics. Taly *et al* [40] propose an operational semantics for a restricted and modified subset of JavaScript.

4.5.4 Comparison With Related Static Analyses of JavaScript

In this section, we give an in-depth overview of the various related works in the are and compare them with our approach. Each method has its advantages and disadvantages and it is useful based on the size of the JavaScript program being evaluated, and also the kind of security properties that need to be verified. Each method has a different way of dealing with `eval` and other dynamic features.

SIF: Chugh *et al* [24] propose a *context-insensitive* and *flow-insensitive* static information flow analysis for JavaScript code called *staged information flow analysis*, SIF. The SIF information flow analysis infers the effects a piece of JavaScript has on the information flows in the program in order to ensure that key confidentiality and integrity properties are not violated.

The information flow policy that needs to be satisfied is specified in the form of sets of pairs of policy elements. A policy element is either a JavaScript program variable or an element called a *hole* (represented as ○), which is a placeholder for `eval`s in the program. A hole represents a piece of code that will be dynamically generated. Each policy pair represents a flow that is disallowed. The example given in the paper checks for two kinds of flow policies. A confidentiality policy requires `(document.cookie, ○)` i.e. the value of `document.cookie` must not flow into any variable within the code generated by an `eval`, since the code can steal this information. An integrity policy requires `(○,document.location)` i.e. the value of `document.location` cannot be modified by dynamic code obtained from an untrusted source.

Given a JavaScript program, with several accesses to the `eval` statement, the SIF static analysis uses a static constraint-based analysis to compute the set of

pairs values that can flow into all known variables that are in the same scope as the hole. These pairs of values could be used to compute two sets (the must not read(MNR) set and the must not write(MNW) set) representing the confidentiality and integrity policy. The MNR set contains all the variables whose value is affected by the protected source variable (`document.cookie` in the above example). The MNW set contains all the variables whose value could affect the protected sink (`document.location` in the above example). The MNR and MNW sets are called the residual policy. If the code in the hole does not write any value to the variables in the MNW set and does not read any values from the MNR set, then the program can be considered to have no vulnerable flows. To ensure this, SIF instruments the `eval`s such that whenever the `eval` is instantiated the code is checked to find if it adheres to the flow policy. The residual policy is recomputed if this newly generated code also has a hole. The authors call this kind of a combination of the static and dynamic analysis staging. Hence the name SIF.

Some of the notable features of SIF are: SIF tracks both implicit and explicit flows, which is required since the threat model is of a malicious code executing on a page. They trust the page author to write the correct confidentiality and integrity policies to use the analysis. `eval`s are not analyzed statically and are treated as holes. To decrease the burden of computing a lot of constraints, they make some simplifications in the contents fo the policy. The policies do not specify general flows like flows between two variables, one policy component should always be a hole. Furthermore, the policies cannot be fine-tuned. All the holes have to adhere to the same policy. The aliasing is very coarse grained. For example, to prevent access to `document.cookie` by an alias like `tmp = document` and accessing `tmp.cookie`, every object's access to the `cookie` property is tainted. This is done because they don't want to taint the `document` object, since it will lead to a lot of false positives. Function bodies are inlined during function calls. All the fields that can be accessed using the `@Proto` field are inserted into the current object. However, this means that fields that are created dynamically will not be inserted into the object. For their evaluation, they only consider evals as holes. To handle dynamically created field names, they unrealistically assume that dynamically created field names are separate from the static field names that are in the program.

Their context insensitive, flow-insensitive analysis is very imprecise and would create a lot of false positives if the approach is used in the analysis of Firefox

extensions. To show this fact, we obtained a copy of SIF and modified it to build a new tool called SIFEX [54], which extends SIF to analyze flows in the Firefox browser extensions. In SIFEX, the constraints for DOM and JavaScript core API are specified. SIFEX analyzed browser extension code for three different kinds of flows: RDF objects to hole, `window.content.document` to hole (hole could be `eval` or `innerHTML`). We used SIFEX to analyze 2452 browser extensions. These extensions had more than 3.8 million lines of JavaScript code. SIFEX found potential vulnerabilities in 169 extensions. The previous version of VEX [26] generates 63 vulnerability reports for the same dataset. Therefore, we can conclude that SIFEX produces more false positives than VEX. We found that many of the false positives are due to the context-insensitivity of SIFEX.

GATEKEEPER: Guarnieri *et al.* [25] propose a mostly-static points-to analysis of JavaScript. The points-to analysis they propose is *context-sensitive* but *flow-insensitive*. The main goal of the points-to analysis is to statically enforce security and reliability policies for JavaScript code.

Their approch is as follows: First, they identify a subset of JavaScript syntax, JavaScript$_{SAFE}$, which is amenable to sound and precise points-to analysis. Certain features of JavaScript like assignment to the `innerHTML` property and dynamic creation of property names for loads and stores are not amenable to static analysis. However, they are an integral part of the target programs that need to be analyzed. Therefore, the authors propose program instrumentation to ensure that these constructs are safe dynamically. The program instrumentation encapsulates these constructs in conditionals such that the construct do not introduce dynamic code. Certain constructs like `eval`, `Function` object constructor, `settimeout`, `setinterval`, and `with` construct are not allowed to be introduced in the code. Second, for the subset of JavaScript, they provide rules to translate each statement in the program representation into a database of facts, expressed in Datalog notation. Using these translation rules, any program written in JavaScript$_{SAFE}$ can be converted into a database of facts, which are basically facts about the program. The rules for the points-to analysis are also given as Datalog facts. Third, they combine these facts with a representation of the native environment of the browser to compute the points-to set of the whole-program.

Finally, they formulate several security and reliability policies terms of Datalog rules which check the points-to sets of the above generated facts for the presence or absence of certain predicates on the points-to facts in the program. The authors

formulate Datalog rules for several such policies including restricting widget capabilities, making sure built-in objects are not modified, preventing code injection attempts, redirect and cross-site scripting detection, preventing global namespace pollution, taint checking, etc.

The primary difference between VEX and GATEKEEPER is that VEX's points-to analysis is specifically tailored towards computing taint information. While GATEKEEPER is a generic points-to analysis for JavaScript. This might be the reason why their taint analysis is very specific. For example, for taint analysis they show a simple case where a tainted variable directly flows into an executable sink. However, they do not seem to consider a scenario where the expression is modified by a function (like a sanitization function) or the value passed to the execution context is a result of a binary operation on two variables, one tainted and the other untainted. It is unclear how to write a policy for such a scenario since the authors don't state explicitly how to deal with binary operations on variables. Additionally, to generate accurate results for the analysis VEX is both context-sensitive and flow-sensitive, while GATEKEEPER is context-sensitive but not flow sensitive. In the paper, the authors conjecture that flow and context sentivity might not make a lot of difference in the precision. However, in our experience with medium sized browser-extensions (≈8000 LOC on average), we found that context sensitivity matters a lot and omitting this would produce a lot of false positives [54]. The absence of flow-sensitivity would make it hard to reason about the presence of sanitization routines in one path in the program. The authors prove that GATEKEEPER's points-to analysis is sound for JavaScript$_{SAFE}$, while we don't have any soundness guarantees for VEX. GATEKEEPER programs are not allowed have `eval`. In contrast, VEX performs a constant-string analysis to find the value that could reach the `eval` function and use this to compute the taint values generated dynamically by `eval`. There are some other subtle differences in how we construct a call-graph and some subtle similarities in how we generate summaries for built-in objects, but these don't affect the analysis specifically.

ENCAP: Taly *et al* [40] provide semantics for a subset of JavaScript, SES$_{light}$, which is used for API confinement. In this paper they also propose a static points-to analysis to prove that any trusted Web sandbox that conforms to the SES$_{light}$ syntax can be verified to be secure i.e. no interleaved sequence of API method calls of a JavaScript sandbox returns a direct reference to a security-critical object.

Similar to GATEKEEPER, they encode the SES$_{light}$ program statements into

Datalog facts. Using this encoding of program statements, the Web sandbox API code is converted into Datalog facts. During this conversion, the tool abstracts heap-locations based on their allocation site. The points-to analysis is also specified in terms of Datalog rules. The analysis is context-insensitive and field-insensitive. It only supports weak updates, which means that they aggregate values with each variable and property assignment. The core JavaScript and DOM functions are also encoded as Datalog rules; different from GATEKEEPER and VEX, which encode them as simplified JavaScript objects. The authors claim that this improves the precision of the analysis. The attacker is also encoded as a set of Datalog rules and facts, whose consequence set is an abstraction of the set of all possible invocations of all the API methods. To test the API confinement, a set of security critical objects that should not be leaked is specified. Once the combined set of Datalog rules and facts reach a fixed-point, confinement can be proved by showing that the security critical objects do not leak to untrusted content. The authors prove that the points-to analysis is sound.

The main difference between ENCAP and VEX is that ENCAP deals with a restricted version of JavaScript which is amenable to static analysis. Specifically, every `eval` in SES_{light} and some object property loads and stores are associated with a set of free variables that could be used dynamically. This information can be used to ensure the soundness of these constructs even though the exact run-time value is not available statically. The language restrictions ensure that the variables are lexically scoped and `this` defines an undefined value. VEX is defined on the full version of JavaScript and therefore it is hard to give soundness guarantees about the analysis. The analysis of ENCAP is context-insensitive and flow-insensitive. This could create a lot of false positives if used to perform an information flow analysis for programs like browser extensions.

An Analytic Framework for JavaScript: Van Horn *et al* [71] propose a reduction semantics for JavaScript which can be used to systematically derive intensional abstract interpretations. In the first step, they transformed the operational semantics of JavaScript propsed by Guha *et al* [38] into an equivalent low level abstract machine called JavaScript Abstract Machine(JAM). Then they derived a systematic abstraction of the entire low-level machine. This abstract machine can be instantiated to obtaint traditional analyses like k-CFA and Configurable Program Analysis(CPA). The taint-analysis proposed in this thesis can also be specified in this framework by associating appropriate abstractions for allocations, loads,

stores, and operations on expressions. It will be interesting to see how this will work in practice.

Acknowledgments This part of the thesis is joint work with Shikhar Agarwal, Samuel T. King, P. Madhusudan, Wyatt Pittman, Nandit Tiku, and Marianne Winslett. We thank Chris Grier and Mike Perry for directing us to the Firefox extension vulnerabilities. We thank the developers of SIF, Ravi Chugh, Jeff Meister, Sorin Lerner, and Ranjit Jhala, who graciously allowed us to modify and use their source code to develop SIFEX. This research was funded in part by NSF CAREER award #0747041, NSF grant CNS #0917229, NSF grant CNS #0831212, grant N0014-09-1-0743 from the Office of Naval Research, and AFOSR MURI grant FA9550-09-01-0539.

Chapter 5

Conclusions

This chapter is organized as follows. Section 5.1 enumerates the contributions of the thesis and the lessons learnt from the work. Section 5.2 concludes with thoughts on the future research directions that this work leads to.

5.1 Conclusions

In this thesis, we described two novel techniques for detection and prevention of two different kinds of injection flaws. These approaches are based on applying information flow analysis techniques to program code. Both these approaches detect the vulnerabilities by inferring the programmers' intention and checking whether maliciously crafted inputs could change the program behaviors to generate an attack.

To detect and prevent SQL injection attacks, we have presented a novel technique, called CANDID, to dynamically deduce the programmer intended structure of SQL queries. We showed how to use CANDID to effectively transform applications so that they guard themselves against SQL injection attacks. The key insight used in CANDID is that *the attack inputs change the intended structure of queries issued*. The basis for CANDID s approach is a powerful idea: a symbolic query computed for a particular execution path can capture the intention of the programmer.

CANDID computed symbolic queries by generating sample benign inputs and forcing them to follow the same path as the user input, thereby producing a sample benign query structure for a particular program path. The program can weed out malicious queries by comparing the sample benign query structure to queries produced from actual user inputs. We implemented CANDID on a test suite that contained seven applications, five of which are commercial, to show strong evidence that CANDID will scale to most web applications. CANDID detected all

the 30 different attack patterns that we tested it with. CANDID imposed a modest performance overhead, ranging from 9% to 43% based on the size of the application and the size of the query generated. We also showed that, compared to the related work, CANDID is more robust in handling sanitization routines, external library calls, and string manipulation functions. CANDID also handles conditional queries (queries whose structure depends on the input value) well. We also show that the benign query structures generated by CANDID are more accurate than other static approaches because it is flow sensitive and dynamic.

At a more abstract level, the idea of computing a symbolic query on sample inputs in order to deduce the intentions of the programmer is a powerful idea that probably has more applications in systems security. There are several approaches in the literature on mining intentions of programmers from code, as such intentions can be used as *specifications* for code, and detection of departure from intentions can be used to infer software vulnerabilities and errors [72, 73, 74]. The idea of using candidate inputs to mine programmer intentions is intriguing and holds much promise.

Our key insight to detect cross-context scripting vulnerabilities in Firefox browser extensions is that *extension vulnerabilities often translate into explicit information flows from injectable sources to executable sinks.* For extensions written with benign intent, most attacks involved the attacker injecting JavaScript into a data item that was subsequently executed by the extension under full browser privileges. In this thesis, we identified key flow patterns of this nature that can lead to security vulnerabilities.

To detect these key flow patterns in the browser extension source code, we developed a novel high-precision static information-flow analysis framework for the JavaScript programming language. The static analysis technique we developed is *context-sensitive*, *field-sensitive* and *flow-sensitive*. We motivated the requirement for high precision static analysis by describing the JavaScript semantics in detail and pointing out the important features of JavaScript that need to be precisely handled to perform an information flow analysis that does not produce a lot of false positives, i.e., declare that a vulnerability may exist, when in fact there is no vulnerability. This analysis has special features to handle the quirks of JavaScript (e.g., it does a constant string analysis for expressions that flow into the `eval` statement that execute dynamically generated code).

Using this new static analysis technique, we developed a tool called VEX, which can be used for detecting security vulnerabilities in Firefox browser exten-

sions. VEX helps in automating the difficult manual process of analyzing browser extensions, by identifying and reasoning about subtle and potentially malicious flows. Experiments on thousands of extensions showed that VEX is successful at identifying flows that indicate potential vulnerabilities. This approach greatly reduced the number of flows that must be vetted manually. Using VEX, we identified 7 previously unknown security vulnerabilities and 5 known vulnerabilities, together with a variety of instances of unsafe programming practices. We also show that VEX is more precise and thereby generates lesser number of false positives when compared to other static analysis techniques, which aim to solve the same problem.

We draw the following conclusions from the thesis:

- Information flow analysis techniques can be used effectively to detect and prevent injection vulnerabilities in web applications. The techniques we developed in this thesis work on the program source code and require no annotation from the programmer. These techniques differ from approaches like model-checking program code, which require the specifications and program code to be written in a special language that can be used for program analysis.

- In case of SQL injection attacks, we believe that the character-level dynamic tainting and the CANDID approach are the only techniques that promise a real scalable automatic solution to dynamically detect and prevent SQL injection attacks. The CANDID approach works particularly well in the case of web applications where all the input is considered to be a string type. Additionally, the SQL query generated has to conform to the SQL grammar and the web application is expecting the input to supply only a part of the SQL query.

- Cross-context script injection attacks are more complicated than the SQL injection attacks. In case of cross-context scripting attacks, the extension takes string inputs from the source. The string inputs could legally be fully formed scripts. These inputs could be malicious or not based on specific API used in the input string. Therefore, it is much more difficult to generate candidate inputs for these attacks. It is very hard to differentiate between legal and illegal inputs based on the structure of the input string. Therefore, we developed VEX to detect cross context scripting vulnerabilities based

on certain attack patterns instead of detecting the attacks dynamically as in CANDID.

- VEX detects the presence of pre-defined flow patterns in the extension code. The presence of flow patterns may not lead to actual attacks. This is possible because of several reasons. One reason could be that the inputs could have been properly sanitized by explicit sanitization routines in the program by sanitization API provided by Firefox, or by implicit sanitization associated with certain JavaScript objects. A second reason could be that the sink objects are not in the privileged chrome context in the browser. Third reason could be that the input was obtained from a trusted web site.

- The above conclusion suggests that a human vetter is required for the several reasons. The first is that there is often a big gap between a vulnerability and an exploit, and it often takes a lot of cleverness and software know-how to move from a vulnerability to an exploit. The second is that the user knows the bigger picture of how the code will be used, e.g., whether it is okay to trust input coming from a particular site. The third is that you had to have a good understanding of browser architecture and functionality to decide which info flows to analyze.

- VEX is a bug-finding tool. The VEX analysis could result in both false-positives (report non-existent flows) and false-negatives (fail to report valid flows). As described in Section 4.3.6, the decision to make a tradeoff between being completely sound versus having no false negatives was motivated by the JavaScript language semantics and our goal to develop a useful tool. Like most static analysis techniques, VEX's precision depends on the precision of the JavaScript summaries created for the core JavaScript objects and the DOM methods. Our experimental results show that VEX is highly successful in detecting attackable vulnerabilities.

5.2 Future Research Directions

5.2.1 Generating Attack Inputs

An interesting future direction is to develop automatic ways to synthesize attacks that exploit injection flaws reported by tools like CANDID and VEX. Using CANDID and VEX, we can deduce the statements in the program which process the inputs from an untrusted source and assign them to an executable sink. In web applications, the inputs are usually strings. Most web applications use a combination of custom API and hand-written sanitization routines to ensure that the strings do not contain malicious scripts and thereby protect the programs from injection attacks.

From our experience of looking at the source code of different Firefox browser extensions, we learned that such sanitization routines can be diverse and hard to write correctly. There is a need for automated techniques to understand the sanitization techniques and ensure that they are able to protect the applications from different kinds of attacks. The two main requirements for the analysis of the sanitization routines are generating string constraints from the source code (including the sanitization routines that work on strings), and solving these string constraints to produce string values that satisfy the constraints. The string thus generated are malicious, then we can conclude that the sanitization routine is not correct.

In the literature, symbolic analysis techniques have been used to generate string constraints from the statements that transform the inputs before they flow into the sinks. *Constraint solving* techniques can be used to generate attack inputs that satisfy the generated string constraints. Several approaches based on string constraint solving have already been proposed for generating test inputs for SQL injection attacks [75, 76, 77, 78]. These approaches differ in the way that they solve the string constraints. The approaches either use either automata based techniques or explicit string constraint solvers on string constraints to generate attack inputs. Most of these approaches use symbolic analysis techniques to generate the string constraints that need to be solved.

Generating attack inputs for JavaScript programs is much more complicated. The JavaScript programming language allows complex string operations. Constraints generated for such string operations are difficult to solve. Some complex string operations that are regularly used in JavaScript programs are:

- Containment in regular expressions/context free grammars.
- Common string manipulation functions like replacement and concatenation.
- JavaScript operations like `substring` and `indexof`, which generate constraints involving the length of strings.

The Kaluza string constraint solver [79] is one of the most recent string constraint solvers for JavaScript. Kaluza generates the string constraints from the path conditions and operations obtained by executing a JavaScript web application. It supports string operations like concatenation and replacement. Further, it supports containment in regular expressions. However, one of the major disadvantages of Kaluza is that it considers strings of only bounded length and hence generation of string constraints for attack inputs is not always possible. Although Kaluza supports a variant of `indexof` and `substring` operations, it requires the fixing of the size values in order to generate constraints. Kaluza obtains these size values by using the Kudzu tool which executes the program on sample inputs and extracts the concrete values. It is not clear how to generate constraints statically in the presence of loops and recursion.

Symbolic analysis is the right step for the analysis of string constraints. However there is a need for tools that generate string constraints statically (without dynamic inputs) and also can solve the constraints robustly without regard to the size of the input and attack strings. There is a requirement for better tools in this area.

5.2.2 Securing Extensible Software

Extensible software is of growing importance and is being increasingly used. Several major browsers, including Firefox, Chrome, IE, and Safari, provide an API for developing browser extensions. With the advent of smart phones like the iPhone and the Android, phone applications are also becoming popular. Other examples of extensible software are social networking applications, and applications that run on the cloud. Increasingly, web applications are moving towards combining data from different sources to provide a better user experience. These applications are written by a few people and downloaded and used by millions of people. These extensions are a big security concern. If compromised, they can be used to

steal users' private information like email, messages, and address book contents. They can also violate the integrity of the devices and databases.

This thesis shows how static program analysis for information flows could be effectively used to find vulnerabilities in browser extensions. A similar kind of analysis can be applied to other extensible applications. The JavaScript analysis techniques developed in this thesis can be used to detect and prevent cross-context scripting attacks and cross-site scripting attacks in other JavaScript based applications, such as Chrome extensions and client-side JavaScript.

One of the reasons for extension vulnerabilities in the Firefox web browser is that the browser architecture is too permissive – it allows every extension to have the same privileges as the browser. Barth *et al* [69] examined the 25 most popular Firefox extensions and found that 88% of them do not need the full set of privileges available to them. Additionally, 76% of the extensions used an unnecessarily powerful API, which made it difficult to automatically reduce their privileges. Based on these experiences, they proposed a new browser extension architecture which had and API for separating access to the different components based on the principles of least privilege and privilege separation. The Google Chrome extension system adopted this new architecture.

Having a good extension architecture is not enough, however. Guha *et al* [70] conducted an analysis of the Chrome extension manifest files and found that this extension model is not entirely effective in limiting the privileges given to the browser extensions. This is partly because the API still provide more privileges than required by the extension. They studied the page access control behavior of 1,139 extensions. They found that only 17% of the extensions correctly limit access to only the web pages that they need to access. The rest of the extensions either use overly permissive wildcard characters to limit page access or request permissions to access all web pages. This discussion strongly suggests that apart from the extension architecture and the APIs provided to the extensions, the access control policy and its implementation in the extension program play a major role in the security of the system. Guha *et al* [70] propose a new framework for authoring, analyzing, verifying, and deploying secure browser extensions.

In conclusion, securing extensible software requires a three pronged approach requiring design of new application architectures (including secure APIs and programming language abstractions), new policy language design, and new analysis techniques. Extensible architectures should be designed with careful thought given to the APIs that are being exposed to the extensions. There is also a need for

new programming languages developed with security as a design goal. These new languages should make authoring and analyzing secure programs an easy process. There is a requirement for new policy languages that are easy to understand and that could be used to describe the different security requirements of the different components of an extensible application. Given the security policies and a program, new analysis techniques should be developed which would efficiently ensure that the program satisfies the security policies.

References

[1] The Open Web Application Security Project (OWASP), "OWASP top 10 web application security risks in year 2010," https://www.owasp.org/index.php/Top_10_2010-Main.

[2] R. S. Liverani, "Cross context scripting with Firefox," April 2010, http://www.security-assessment.com/files/documents/whitepapers/Cross_Context_Scripting_with_Firefox.pdf.

[3] K. J. Biba, "Integrity considerations for secure computer systems," USAF Electronic Systems Division, Bedford, MA, Tech. Rep. ESD-TR-76-372, Apr. 1977.

[4] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003.

[5] C. Anley, "Advanced SQL injection in SQL server applications, White paper, Next Generation Security Software Ltd." Tech. Rep., 2002.

[6] O. Maor and A. Shulman, "SQL injection signatures evasion. White paper, Imperva," Tech. Rep., 2002. [Online]. Available: http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html

[7] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th USENIX Security Symposium*, July 2006, pp. 179–192.

[8] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*. Berkeley, CA, USA: USENIX Association, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1251398.1251416 pp. 18–18.

[9] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," 2005.

[10] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *20th IFIP International Information Security Conference*, 2005.

[11] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," 2006.

[12] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," January 2006.

[13] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," in *Proc. of the International Symposium on Secure Software Engineering*, Mar. 2006.

[14] W. Halfond and A. Orso, "AMNESIA: Analysis and monitoring for neutralizing SQL-injection," 2005.

[15] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of SQL attacks," in *Proceedings of the Conference on Detection of Intrusions and Malware Vulnerability Assessment*, 2005.

[16] W. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proceedings of the Twelfth ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2006.

[17] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," in *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, 2005.

[18] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: Preventing SQL Injection Attacks Using Dynamic Candidate Evaluations," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2007, pp. 12–24.

[19] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *POPL '07*. New York, NY, USA: ACM, 2007, pp. 237–249.

[20] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov, "Javascript instrumentation in practice," in *APLAS '08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 326–341.

[21] M. Dhawan and V. Ganapathy, "Analyzing information flow in javascript-based browser extensions," in *ACSAC'09: Proceedings of the 25th Annual Computer Security Applications Conference*. Honolulu, Hawaii, USA: IEEE Computer Society Press, Los Alamitos, California, USA, December 2009, http://doi.ieeecomputersociety.org/10.1109/ACSAC.2009.43. pp. 382–391.

[22] V. Djeric and A. Goel, "Securing script-based extensibility in web browsers," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1929820.1929851 pp. 355–370.

[23] M. Ter Louw, J. S. Lim, and V. N. Venkatakrishnan, "Extensible web browser security," Lucerne, Switzerland, 2007.

[24] R. Chugh, J. Meister, S. Lerner, and R. Jhala, "Staged information flow for javascript," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, M. Hind and A. Diwan, Eds., no. 978-1-60558-392-1. Dublin, Ireland: ACM, June 15-21 2009, pp. 50–62.

[25] S. Guarnieri and B. Livshits, "Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code," in *Proceedings of USENIX Security '09*, 2009, pp. 151–168.

[26] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "Vex: vetting browser extensions for security vulnerabilities," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1929820.1929850 pp. 22–22.

[27] K. K. Mookhey and N. Burghate, "Detection of SQL Injection and Cross-site Scripting Attacks." [Online]. Available: http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks

[28] RSnake, "XSS Cheat Sheet." [Online]. Available: http://ha.ckers.org/xss.html

[29] January 16 2005. [Online]. Available: http://packetstormsecurity.org/UNIX/misc/fuzzer-1.1.tar.gz

[30] J. Ruderman, "Introducing jsfunfuzz." [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[31] [Online]. Available: http://www.businessinfo.co.uk/labs/jsfuzz/fuzz.php

[32] J. Adriaans and P. Laguna, "HackBar." [Online]. Available: https://addons.mozilla.org/en-US/firefox/addon/hackbar/

[33] Security Compass, "XSS Me." [Online]. Available: https://addons.mozilla.org/en-US/firefox/addon/xss-me/

[34] Security Compass, "SQL Inject Me." [Online]. Available: https://addons.mozilla.org/en-US/firefox/addon/sql-inject-me/

[35] Francois Larouche, "SQL Power Injector." [Online]. Available: http://www.sqlpowerinjector.com/

[36] D. Flanagan, *JavaScript: The Definitive Guide*, 4th ed. O'Reilly Media, 2001, no. 0-596-00048-0.

[37] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Ibm javascript security test suite." [Online]. Available: https://researcher.ibm.com/researcher/view_page.php?id=1598

[38] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," in *Proceedings of the 24th European conference on Object-oriented programming*, ser. ECOOP'10. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1883978.1883988 pp. 126–150.

[39] S. Maffeis, J. Mitchell, and A. Taly, "An operational semantics for JavaScript," in *Proc. of APLAS'08*, ser. LNCS, vol. 5356, 2008, pp. 307–325.

[40] A. Taly, U. Erlingsson, M. S. Miller, J. C. Mitchell, and J. Nagra, "Automated analysis of security-critical javascript apis," in *IEEE Security & Privacy (Oakland conference)*, 2011.

[41] J. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "Adsafety: Type-based verification of javascript sandboxing," in *USENIX Security*, 2011.

[42] C. Saftoiu, "Jstrace: Run-time type discovery for javascript," May 2010.

[43] P. Thiemann, "Towards a type system for analyzing javascript programs," in *European Symposium on Programming*, 2005, pp. 408–422.

[44] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Proc. 16th International Static Analysis Symposium (SAS)*, ser. LNCS, vol. 5673. Springer-Verlag, August 2009.

[45] C. Anderson and P. Giannini, "Type checking for javascript," *Electronic Notes in Theoretical Computer Science*, vol. 138, pp. 37–58, 2005.

[46] B. Weber, "Type safety for javascript."

[47] A. Guha, C. Saftoiu, and S. Krishnamurthi, "Typing local control and state using flow analysis," in *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ser. ESOP'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?id=1987211.1987225 pp. 256–275.

[48] B. Livshits and S. Guarnieri, "Gulfstream: Incremental static analysis for streaming javascript applications," 2010.

[49] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for ajax intrusion detection," in *Proceedings of the 18th international conference on World wide web*, ser. WWW '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1526709.1526785 pp. 561–570.

[50] S. Maffeis, J. Mitchell, and A. Taly, "Isolating javascript with filters, rewriting, and wrappers," in *Proc of ESORICS'09*. LNCS, 2009.

[51] S. Maffeis and A. Taly, "Language-based Isolation of Untrusted JavaScript," in *In Proceedings of Computer Security Foundations*, ser. IEEE, 2009, See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[52] S. Maffeis, J. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *Proc of IEEE Security and Privacy'10*. IEEE, 2010.

[53] S. Maffeis, J. Mitchell, and A. Taly, "Run-time enforcement of secure javascript subsets," in *Proc of W2SP'09*. IEEE, 2009.

[54] S. Agarwal, S. Bandhakavi, and M. Winslett, "SIFEX: Tool for static analysis of browser extensions for security vulnerabilities," Poster at Annual Computer Security Applications Conference, December 2010.

[55] "Soot: a java optimization framework," http://www.sable.mcgill.ca/soot/.

[56] "Online SQL syntax checker," http://www.wangz.net/gsqlparser/sqlpp/sqlformat.htm.

[57] Apache, "The JMeter project," http://jakarta.apache.org/jmeter/.

[58] R. A. McClure and I. H. Kruger, "Sql dom: compile time checking of dynamic sql statements," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005.

[59] W. Cook and S. Rai, "Safe query objects: Statically-typed objects as remotely-executable queries," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005.

[60] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks," 2004, pp. 292–302.

[61] IAOSS, "NoScript Firefox extension," http://noscript.net/.

[62] A. Boodman, "The Greasemonkey Firefox extension," https://addons.mozilla.org/en-US/firefox/addon/748.

[63] R. S. Liverani and N. Freeman, "Abusing Firefox extensions, Defcon 17," July 2009.

[64] C. Waterson, "RDF in fifty words or less," https://developer.mozilla.org/en/RDF_in_Fifty_Words_or_Less.

[65] "ANTLR Parser Generator," http://www.antlr.org.

[66] N. Freeman and R. S. Liverani, "Exploiting cross context scripting vulnerabilities in Firefox," April 2010, http://www.security-assessment.com/files/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf.

[67] J. Verdurmen, "Firefox extension security," Radboud University Nijmegen, The Netherlands, 2008, Undergraduate Thesis.

[68] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan, "Enhancing web browser security against malware extensions," in *Journal in Computer Virology*, vol. 4, no. 3, 2008, pp. 179–195.

[69] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities," in *Network and Distributed System Security Symposium (NDSS)*, vol. 17, February 2010.

[70] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified security for browser extensions," in *Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, California, USA, May 2011.

[71] D. V. Horn and M. Might, "An analytic framework for javascript," 2011, in submission.

[72] G. Ammons, R. Bodik, and J. Larus, "Mining specifications," 2002.

[73] R. Alur, P. Cerny, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," in *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

[74] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection." in *TACAS*, 2005, pp. 461–476.

[75] F. Yu, M. Alkhalaf, and T. Bultan, "Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses," in *ASE*, 2009, pp. 605–609.

[76] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *ISSTA*, 2009, pp. 105–116.

[77] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium, SAS '03*, ser. LNCS, vol. 2694. Springer-Verlag, June 2003, available from `http://www.brics.dk/JSA/`. pp. 1–18.

[78] P. Hooimeijer and W. Weimer, "Solving string constraints lazily," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010.

[79] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.