Systematic Techniques for Finding and Preventing Script Injection Vulnerabilities

Prateek Saxena

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-170
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-170.html

June 29, 2012

Copyright © 2012, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Systematic Techniques for Finding and Preventing Script Injection Vulnerabilities

by

Prateek Saxena

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor David Wagner
Professor Brian Carver

Fall 2012

Systematic Techniques for Finding and Preventing Script Injection Vulnerabilities

Copyright 2012
by
Prateek Saxena

Abstract

Systematic Techniques for Finding and Preventing Script Injection Vulnerabilities

by

Prateek Saxena

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

Computer users trust web applications to protect their financial transactions and online identities from attacks by cyber criminals. However, web applications today are riddled with security flaws which can compromise the security of their web sessions. In this thesis, we address the problem of automatically finding and preventing script injection vulnerabilities, one of the most prominent classes of web application vulnerabilities at present. Specifically, this thesis makes three contributions towards addressing script injection vulnerabilities. First, we propose two techniques that together automatically uncover script injection vulnerabilities in client-side JavaScript components of web applications without raising false positives. Second, we empirically study the use of sanitization, which is the predominant defense technique to prevent these attacks today. We expose two new classes of errors in the practical use of sanitization in shipping web applications and demonstrate weaknesses of emerging defenses employed in widely used web application frameworks. Third, we propose a type-based approach to automatically perform correct sanitization for applications authored in emerging web application frameworks. Finally, we propose a conceptual framework for a sanitization-free defense against script injection vulnerabilities, which can form a robust second line of defense.

To my parents, Krati and my brother Siddharth.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis is a result of ideas that were born out of discussions and collaboration with many people. Without them this work would not be possible. I am responsible for any shortcomings that remain in this thesis.

First, I thank my adviser and thesis committee chair Prof. Dawn Song. Her insights and feedback have directly shaped the technical ideas in this thesis. But, more importantly, her passion for scientific research is contagious and has had an indelible effect on my personality. My other committee members, Prof. David Wagner and Prof. Brian Carver, have provided valuable feedback on the thesis. Thanks to Prof. David Wagner for insightful comments on papers that are part of this thesis; his words of encouragement and guidance have helped me throughout my PhD. Thanks to Prof. Brian Carver for suggestions on improving this manuscript. I am also indebted to Prof. R. Sekar who convinced me to pursue a research career.

My colleagues made research and fun inseparable. Many thanks to Adam Barth, Stephen McCamant, Pongsin Poosankam, Chia Yuan Cho, Steve Hanna, Joel Weinberger, Noah M. Johnson, Kevin Zhijie Chen, Adrienne Felt, Matt Finifter, Devdatta Akhawe, Adrian Mettler and Juan Caballero for discussions and feedback on this work.

I have learned broadly from my mentors and collaborators David Molnar, Ben Livshits, Patrice Godefroid, Margus Veanes and various team members during the work done at Microsoft Research. I enjoyed working with Mike Samuel; his perspectives and effort were instrumental in making some of our ideas practical at Google. Thanks to Vijay Ganesh and Adam Kiezun for their help on the HAMPI string solver.

I have never had to look far for sources of constant inspiration and encouragement. My wife, Krati, walked every step of the way sporting a disarming smile; my journey couldn't be easier. I am indebted to my Mom for her unconditional love; my Dad for his undying spirit and for being an aspiring entrepreneur who I can only hope to emulate; and finally, my brother Siddharth who is a real-life proof of what tenacity can achieve. Finally, thanks to my friends (you know who you are) for unforgettable support at times when things seemed low—you have all made contributions to this work.

Chapter 1

Introduction

The web is our primary gateway to many critical services and offers a powerful platform for emerging applications. As the underlying execution platform for web applications grows in importance, its security has become a major concern. Web application vulnerabilities have become pervasive in web applications today, yet techniques for finding and defending against them are limited. How can we build a secure web application platform for the future? In this thesis, we answer this research question in part. We tackle the problem of developing techniques to automatically find and prevent *script injection* (or *scripting*) vulnerabilities—a class of web vulnerabilities permissive in web applications today.

Web languages, such as HTML, have evolved from light-weight mechanisms for static data markup to full-blown vehicles for supporting dynamic execution of web application logic. HTML allows inline constructs both to embed untrusted data and to invoke code in higher-order languages such as JavaScript. Web applications often embed data controlled by untrusted adversaries inline within the HTML code of the web application. For example, a blogging application often embeds untrusted user comments inline within the HTML content of the blog. HTML and other web languages lack principled mechanisms to separate trusted code from inline data and to further isolate untrusted data (such as user-generated content) from trusted application data. Script injection vulnerabilities arise when untrusted data controlled by an adversary is interpreted by the web browser as trusted application (script) code. This causes an attacker to gain higher privileges than intended by the web application, typically granting untrusted data the same authority as the web application's code. Well-known example categories of such attacks are *cross-site scripting* (or XSS) [94] and *cross-channel scripting* (or XCS) [18] attacks.

Scripting vulnerabilities are highly pervasive and have been recognized as a prominent category of computer security vulnerabilities. Software errors that result in script injection attacks are presently rated as the fourth most serious of software errors in the CWE's Top 25 list for the year 2011 [31]. OWASP's Top 10 vulnerabilities ranks scripting attacks as the second most dangerous of web vulnerabilities in 2010 [89]. Web Application Security Consortium's XSS vulnerability report shows that over 30% of the web sites analyzed in 2007 were vulnerable to XSS attacks [123]. In addition, there exist publicly available repositories

of real-world XSS vulnerabilities, which have 45517 reported XSS vulnerabilities (as of June 10, 2012) with new ones being added constantly [130].

Most prior research on finding scripting vulnerabilities has focused on server-side components [8, 132, 16, 62, 85, 75, 107, 121, 72]. In this thesis, we focus on analysis of scripting vulnerabilities in client-side code written in JavaScript, which has received little attention prior to our research. In contrast to several concurrent works have investigated static analysis approaches to analyzing JavaScript, our work employs dynamic analysis techniques. Our aim is to develop techniques which have no false positives and which produce witness exploit inputs when they uncover a vulnerability.

Several mechanisms have been discussed, both in practice and in research, on preventing scripting vulnerabilities. In this thesis, we explore two directions towards preventing script injection vulnerabilities in web applications. First, we investigate the most predominant prevention technique that developers employ in practice and explain the challenges in getting it right. We then propose techniques to automate this defense, thereby shifting the burden of applying correct prevention measures from the developers to the underlying compilation tools. Second, we investigate alternative architecture for web applications that preserves a strict separation between untrusted data and application code during the parsing operations of the browser. This architecture obviates the need for today's predominant prevention techniques which are notoriously error-prone.

1.1 Contributions

This thesis makes the following contributions:

Automatic Techniques for Finding Scripting Vulnerabilities. We propose two white-box analysis techniques for finding scripting vulnerabilities and build the first systems to apply these techniques to JavaScript applications. Our techniques are based on dynamic analysis and find vulnerabilities in several real-world applications *without raising false positives*. The first of these techniques is called taint-enhanced black-box fuzzing (**Chapter 3**). It improves over prior black-box dynamic fuzzing approaches by combining it with a previous white-box analysis called dynamic taint analysis [86]. Our second technique improves over white-box-based dynamic testing methods, specifically dynamic symbolic execution [40], by introducing a more comprehensive symbolic reasoning of strings (**Chapter 4**).

Analysis of Existing Defenses. In this thesis, we identify implicit assumptions underlying today's deployed prevention techniques (**Chapter 5**). Specifically, we explain the assumptions and subtleties underlying *sanitization*, the predominant defense technique deployed in practice. Our empirical analysis of large-scale applications uncovers two new classes of errors in sanitization practices when implemented manually by developers. Furthermore, we outline several incompleteness in mechanisms implemented by emerging web application frameworks that try to enforce sanitization defenses automatically.

Techniques for Preventing Scripting Vulnerabilities. We propose a type system and inference engine to address the problem of automatic sanitization (or *auto-sanitization*), which eliminates the error-prone practice of manually applying sanitization in web applications (**Chapter 6**). Together, with external techniques to identify all untrusted variables [97, 132] and to implement correct sanitizers [51], this work provides a basis to achieve correct sanitization-based defense in emerging web applications automatically.

We propose a second defense mechanism which eliminates the need for sanitization-based defense (**Chapter 7**). In this work, we also develop a *sanitization-free* architecture for web applications, relying on a collaboration between the server and the client web browser. We introduce a fundamental integrity property called *document structure integrity* and sketch mechanisms to enforce it during the end-to-end execution of the application. Our work demonstrates an initial proof-of-concept for implementing these in existing applications with minimal impact to backwards compatibility.

1.2 Statement of Joint Work

The development of techniques and systems presented in Chapter 3 and Chapter 4 was led by Prateek Saxena. In addition to Prateek Saxena, contributors to the work presented in Chapter 3 include Steve Hanna, Pongsin Poosankam and Dawn Song. Contributors in addition to Prateek Saxena for the work presented in Chapter 4 include Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant and Dawn Song.

Analysis of sanitization use large-scale applications presented in Chapter 5 was led by Prateek Saxena. Contributors in addition to Prateek Saxena to this work include David Molnar and Ben Livshits. The rest of the work presented in Chapter 5 was joint work between Prateek Saxena, Joel Weinberger, Devdatta Akhawe, Matthew Finifter, Richard Shin and Dawn Song.

The development of auto-sanitization technique presented in Chapter 6 was joint work between Prateek Saxena, Mike Samuel and Dawn Song. Contributors to the sanitization-free defense technique proposed in Chapter 7 in addition to Prateek Saxena include Yacin Nadji and Dawn Song.

Chapter 2

Background & Overview

Web applications are distributed applications consisting of components that execute either on a web server or on a user's client. The code which executes on the server, which we refer to as *server-side* code, is usually written in languages such as Java, C/C++, PHP, ASP.NET. It is responsible for processing the HTTP inputs and operates on data stored in the server-side database or file-system. In response to a HTTP request, the server-side code sends an HTTP response which consists of additional *client-side* code. Client-side code consists of languages parsed and executed by the browser such as HTML, CSS and JavaScript. Client- and server-side components communicate with each other over the network, typically over custom protocols layered on HTTP.

In this thesis, we focus on building techniques to find and prevent script injection vulnerabilities, one of the most prominent vulnerabilities affecting web applications today. These vulnerabilities affect both client- and server-side components of web applications. We provide some examples of these vulnerabilities and define preliminary terminology used throughout this thesis in Section 2.1.

2.1 Script Injection Vulnerabilities: Definition & Examples

Scripting vulnerabilities arise when content controlled by an adversary (referred to as *untrusted data*) flows into critical operations of the program (referred to as *critical sinks*) without sufficient security checks. When untrusted data is parsed or evaluated as trusted code by the web browser, a scripting attack results. This causes an attacker to gain higher privileges than intended by the web application, typically granting untrusted data the same privileges as the web application's code. Well-known example categories of such code-injection attacks include cross-site scripting [94] and cross-channel scripting [18] attacks.

The definitions of critical sinks and untrusted data inputs are application-specific. The intended security policy for certain applications permit data taken from users or third-party web sites to be evaluated as script code. On the other hand, many other applications do not

```
 1:   <body>
 2:   <div id='WelcomeMess'> Welcome! </div>
 3:   <div id='$GET['FriendID-Status']' name='status'> </div>
13:   </body>
```

Figure 2.1: A snippet of HTML pseudo-code generated by a social networking application server vulnerable to scripting attack. Untrusted input data, identified by the `$GET['...']` variables, to the server is echoed inline in the HTML response and without any modification or sanitization.

intend untrusted data inputs (such as user-generated content) to be executed by the browser as code. Our techniques assume that such a security specification is externally provided.

Server-side Script Injection Vulnerabilities

Script injection attacks in server-side applications have been investigated in depth by prior work. We provide an example of a typical scripting attack for exposition.

In this example, all the HTTP data inputs to the web application server are treated as untrusted data. In this application, the security policy forbids untrusted data to be executed as scripts or HTML markup when processed by the web browser. A script injection vulnerability is one that allows injection of untrusted data into a victim web page which is subsequently interpreted in a malicious way by the browser on behalf of the victim web site.

An Example. We show a hypothetical example of HTML code that a buggy web application emits in Figure 2.1. Places where untrusted user data is inlined are denoted by elements of `$GET['...']` array (signifying data directly copied from GET/POST request parameters). In this example, the server expects the value of `$GET['MainUser']` to contain the name of the current user logged into the site, and `$GET['FriendID-Status']` to contain a string with the name of another user and his status message ("online" or "offline") separated by a delimiter ("="). If the untrusted input data is a malicious string, such as `' onmouseover=javascript:bad()`, a script injection attack occurs. In this attack, the malicious value of `$GET['FriendID-Status']` prematurely closes the `id` attribute of the `<div>` tag on line 3 and injects unintended HTML attributes and/or tags. This particular attack string closes the string delimited by the single quote character, which allows the attacker to inject a JavaScript attribute called `onmouseover`. The value of the injected JavaScript attribute executes as arbitrary JavaScript code which we depict as a function call `bad()`. This exploit string is one of many possible vectors that is publicly known—over 200 such vectors are available online [94]. This attack example belongs to a sub-class of script injection attacks commonly referred to as a reflected cross-site scripting attack.

Client-side Script Injection Vulnerabilities

Much prior vulnerability research has focused primarily on the server-side components of web applications. Scripting vulnerabilities can arise in client-side components, such as those written in JavaScript, as well [68]. We present examples of *client-side script injection* vulnerabilities, a subclass of scripting vulnerabilities which result from bugs in the client-side code. In a client-side script injection vulnerability, critical sinks are operations in the client-side code where data is used with special privilege, such as in a code evaluation construct.

Client-side script injection vulnerabilities are different from server-side scripting vulnerabilities in a few ways. For example, one type of client-side script injection vulnerability involves data that enters the application through the browser's cross-window communication abstractions and is processed completely by JavaScript code, without ever being sent back to the web server. Another type of client-side script injection vulnerability is one where a web application server sanitizes untrusted data sufficiently before embedding it in its initial HTML response, but does not sanitize the data sufficiently for its use in the JavaScript component.

Client-side script injection vulnerabilities are becoming increasingly common due to the growing complexity of JavaScript applications. Increasing demand for interactive performance of rich web 2.0 applications has led to rapid deployment of application logic as client-side scripts. A significant fraction of the data processing in AJAX applications (such as Gmail, Google Docs, and Facebook) is done by JavaScript components. JavaScript has several dynamic features for code evaluation and is highly permissive in allowing code and data to be inter-mixed. As a result, attacks resulting from client-side script injection vulnerabilities often result in compromise of the web application's integrity.

In the security policy of many web applications, any data which is controlled by an external (or third-party) web origin is treated as untrusted data. Additionally, user data (such as content of GUI form fields or `textarea` elements) is treated as untrusted. Untrusted data could enter the client-side code of a web application in three ways. First, data from an untrusted web attacker could be reflected in the honest web server's HTML response and subsequently read for processing by the client side code. Second, untrusted data from other web sites could be injected via the cross-window communication interfaces provided by the web browser. These interfaces include `postMessage`, URL fragment identifiers, and window/frame cross-domain properties. Finally, user data fed in through form fields and text areas is also marked as untrusted.

The first two untrusted sources are concerned with the threat model where the attacker is a remote entity that has knowledge of a client-side script injection vulnerability in an honest (but buggy) web application. The attacker's goal is to remotely exploit a client-side script injection vulnerability to execute arbitrary code. The attack typically only involves enticing the user into clicking a link of the attacker's choice (such as in a reflected XSS attack).

We also consider the "user-as-an-attacker" threat model where the user data is treated as untrusted. In general, user data should not be interpreted as web application code. For instance, if user can inject scripts into the application, such a bug can be used in conjunction

```
1: var chatURL = "http://www.example.com/";
2: chatURL += "chat_child.html";
3: var popup = window.open(chatURL);
4: ...
5: function sendChatData (msg) {
6:   var StrData = "{\"username\": \"joe\", \"message\": \"" + msg + "\"}";
7:   popup.postMessage(StrData, chatURL);
}
```

Figure 2.2: An example of a chat application's JavaScript code for the main window, which fetches messages from the backend server at `http://example.com/`

with other vulnerabilities (such as login-CSRF vulnerabilities) in which the victim user is logged-in as the attacker while the application behavior is under the attacker's control [12]. In our view, these vulnerabilities can be dangerous as they allow sensitive data exfiltration, even though the severity of the resulting exploits varies significantly from application to application.

Running Example. For exposition, we introduce a running example of a hypothetical AJAX chat application here which we will revisit in Chapter 3. The example application consists of two windows. The main window, shown in Figure 2.2, asynchronously fetches chat messages from the backend server. Another window receives these messages from the main window and displays them, the code for which is shown in Figure 2.3. The communication between the two windows is layered on `postMessage`[1], which is a string-based message passing mechanism included in HTML 5. The application code in the display window has two sources of untrusted data—the data received via `postMessage` could be sent by any browser window, and the `event.origin` property, which is the origin (port, protocol and domain) of the sender.

We discuss some of the attacks possible from exploiting the errors in this example application code next. The script injection attack is discussed first and then we outline three other related attacks which are less severe but problematic because they cause escalation of privileges afforded by the remote attacker.

- *Script injection.* Script injection is possible because JavaScript can dynamically evaluate both HTML and script code using various DOM methods (such as `document.write`) as well as JavaScript native constructs (such as `eval`). This class of attacks is commonly referred to as DOM-based XSS [68]. An example of this attack is shown in Figure 2.3 on line 19. In the example, the display child window uses `eval` to serialize the input string from a JSON format, without validating for its expected structure. Such attacks are prevalent today because popular data exchange interfaces, such as JSON, were specifically designed for use with the `eval` constructs. In Section 3.4, we outline

[1]In the `postMessage` interface design, the browser is responsible for attributing each message with the domain, port, and protocol of the sender principal and making it available as the "origin" string property of the message event [13, 119]

```
1:function ParseOriginURL (url) {
2: var re=/(.*?):\/\/(.*?)\.com/;
3: var matches = re.exec(url);
4: return matches;
5:}
6:
7:function ValidateOriginURL (matches)
8:{
9: if(!matches) return false;
10: if(!/https?/.test(matches[1]))
11:        return false;
12: var checkDomRegExp = /example/;
13: if(!checkDomRegExp.test (matches[2])) {
14:        return false;  }
15: return true;     // All Checks Ok
16:}
17:// Parse JSON into an array object
18:function ParseData (DataStr) {
19:  eval (DataStr);
20:}
21:function receiveMessage(event) {
22: var O = ParseOriginURL(event.origin);
23:  if (ValidateOriginURL (O)) {
24:   var DataStr = 'var new_msg =(' +
25:                 event.data + ');';
26:   ParseData(DataStr);
27:   display_message(new_msg);
29:   var backserv = new XMLHttpRequest(); ...;
30:   backserv.open("GET","http://example.com/srv.php?
      call=confirmrcv&msg="+new_msg["message"]);
31:   backserv.send();} ... } ...
32: window.addEventListener("message",
               receiveMessage,...);
```

Figure 2.3: An example vulnerable chat application's JavaScript code for a child message display window, which takes chat messages from the main window via `postMessage`. The vulnerable child message window code processes the received message in four steps, as shown in the `receiveMessage` function. First, it parses the principal domain of the message sender. Next, it tries to check if the origin's port and domain are "http" or "https" and "example.com" respectively. If the checks succeed, the popup parses the JSON [58] string data into an array object and finally, invokes a function for displaying received messages. In lines 29-31, the child window sends confirmation of the message reception to a backend server script.

additional phishing attacks in iGoogle gadgets layered on such XSS vulnerabilities, to illustrate that a wide range of nefarious goals can be achieved once the application integrity is compromised.

- *Origin Mis-attribution.* Certain cross-domain communication primitives, such as via `postMessage`, are designed to facilitate sender authentication. Applications using `postMessage` are responsible for validating the authenticity of the domain sending the message. The example in Figure 2.3 illustrates such an attack on line 13. The vulnerability arises because the application checks the domain field of the origin parameter insufficiently, though the protocol sub-field is correctly validated. The failed check allows any domain name containing "example", including an attacker's domain hosted at "evilexample.com", to send messages. As a result, the vulnerable code naively trusts the received data even though the data is controlled by an untrusted principal. In the running example, for instance, an untrusted attacker can send chat messages to victim users on behalf of benign users.

- *HTTP Parameter Polution.* Many AJAX applications use untrusted data to construct URL parameters dynamically, which are then used to make HTTP requests (via `XMLHttpRequest`) to a backend server. Several of these URL parameters play the role of application-specific commands in these HTTP requests. For instance, the chat application in the example sends a confirmation command to a backend script on lines 29-31. The backend server script may take other application commands (such as adding friends, creating a chat room, and deleting history) similarly from HTTP URL parameters. If the HTTP request URL is dynamically constructed by the application in JavaScript code (as done on line 30) using untrusted data without validation, the attacker could inject new application commands by inserting extra URL parameters. These attacks are called HTTP parameter pollution attacks [7]. Since the victim user is already authenticated, parameter pollution allows the attacker to perform unintended actions on behalf of the user. For instance, the attacker could send `hi&call=addfriend&name=evil` as the message which could result in adding the attacker to the buddy list of the victim user.

- *Cookie-sink vulnerabilities.* Web applications often use cookies to store session data, user's history and personal preference settings. These cookies may be updated and used in the client-side code. If an attacker can control the value written to a cookie, it may fix the values of the session identifiers (which may result in a session fixation attack) or corrupt the user's preferences and history data.

2.2 Techniques for Finding Script Injection Vulnerabilities Automatically

If we can develop techniques to automatically find script injection vulnerabilities in web applications, it is possible to eliminate many exploitable vulnerabilities before applications are used in deployment. Analysis techniques for finding scripting vulnerabilities, especially in server-side code, have been widely researched [132, 72, 71, 18, 127, 62, 55, 87, 75, 8]. In this thesis, we focus on techniques for finding these vulnerabilities in JavaScript code, which have received much lesser attention in research prior to our work. Prior techniques for finding scripting vulnerabilities in web applications largely fall into 3 categories: manual analysis, static analysis and dynamic analysis. We discuss these ideas below, outline the new challenges posed by JavaScript and explain how our techniques are different at a high-level. Our techniques aim to uncover vulnerabilities without raising false positives by constructing concrete exploit inputs automatically for vulnerabilities found. To achieve this, our techniques explore the program behavior systematically and reason about transformation (such as *validation* or *sanitization* operations) that the application may perform on the untrusted data.

Differences from Existing Techniques. Fuzzing or black-box testing is a popular lightweight mechanism for testing applications. However, black-box fuzzing does not scale well with a large number of inputs and is often inefficient in exploration of the application's path space. A more directed approach used in the past in the context of server-side code analysis is based on *dynamic taint-tracking* [132]. Dynamic taint analysis is useful for identifying a flow of data from an untrusted source to a critical operation. However, dynamic taint-tracking alone alone can not determine if the application sufficiently validates untrusted data before using it. Consider a canonical example of an application logic that transforms the dangerous text `<script>` to an empty string, if it appears in the input. Dynamic taint will only identify the characters in the output of this operation that were not replaced by the constant empty string, and only so if the dynamic values on the given execution contain the text string `<script>`. Dynamic taint tracking does not capture the full behavior of the sanitization logic under different inputs. To overcome this limitation in practice, dynamic taint analysis tools such as PHPTaint [117] use the strategy of pre-identifying certain operations as validation or sanitization constructs. However, when parsing operations and validation checks are syntactically indistinguishable from each other in application code, such pre-specification is not possible and the alternative approximations are difficult. If an analysis tool treats all string operations on the input as parsing constructs, it will fail to identify validation checks and will report false positives even for legitimate uses (as shown by our experiments in Section 4.6). On the other hand, if the analysis treats any use of untrusted data which has been passed through a parsing/validation construct as safe, it is likely to miss many bugs. Static analysis is another approach [47, 26, 9, 48, 74]; however static analysis tools do not directly provide concrete exploit instances and require additional developer analysis to prune away false positives.

We aim to bridge the shortcomings of these techniques. We aim to develop techniques that can explore the program's functionality systematically and generate concrete witness inputs that demonstrate an exploit. In this regard, symbolic execution techniques have been used for discovering and diagnosing vulnerabilities in server-side logic [66, 17, 59, 121]. However, web applications pervasively use complicated operations on string and arrays data types, both of which raise difficulties for decision procedures involved in symbolic execution techniques. The power and expressiveness of string decision procedures prior to our work was limited. Practical implementations of string decision procedures prior to our work did not deal with the generality of JavaScript string constraints involving common operations (such as `String.replace`, regular expression match, concatenation and equality) expressed together over multi-variable, variable-length inputs [66, 17, 59, 50]. Other symbolic analyses have been limited to a subset of input-transformation operations in PHP [8]. These limitations of prior symbolic execution tools motivate the need for our solutions.

Challenges in JavaScript Code Analysis. The first challenge of holistic application analysis is in dealing with the complexity of JavaScript. Many JavaScript programs use code evaluation constructs to dynamically generate code as well as to de-serialize strings into complex data structures (such as JSON arrays/objects). In addition, the language supports myriad high-level operations on complex data types. This makes the task of practical analysis, especially based on static analysis methods, difficult.

In JavaScript application code, we observe that parsing operations are syntactically indistinguishable from validation checks. This makes it infeasible for automated syntactic analyses to reason about the sufficiency of validation checks in isolation from the rest of the logic. Due to the convenience of their use in the language, developers tend to treat strings as a universal type for exchanging both code and data values. Consequently, complex string operations such as regular expression match and replace are pervasively used both for parsing input and for performing custom validation checks.

Sub-challenges & Our Approach. The problem of finding vulnerabilities in JavaScript-heavy applications has two orthogonal sub-challenges— (a) automatically exploring the execution space of client-side JavaScript code, and, (b) and finding an input that exposes a vulnerability in some explored program path. We decouple our analysis into two orthogonal sub-analyses to address these two sub-challenges:

- *Single-path Analysis.* First, we develop a *single-path analysis* technique which focuses on finding an exploit input that traverses a given path in the program. Specifically, the input to a single-path analysis is an initial benign test case that executes some path in the program. The analysis aims to find an exploit instance by systematically searching the equivalence class of inputs that force the program execution down the same path as the given benign input. We present this technique and our Flax tool that implements it in Chapter 3.

- *Multiple Path Analysis.* Second, we develop a system called Kudzu that automatically explores the execution space of client-side JavaScript code. This technique employed in

KUDZU takes as input an initial test case and synthesizes a larger harness of test cases that explore the program execution space in more depth. To explore the application's behavior, our techniques utilizes dynamic symbolic execution on JavaScript code with deeper modeling of string operations. In addition, it combines symbolic execution with automatic GUI exploration to explore the space of input events to the application. We present details of our technique and KUDZU in Chapter 4.

2.3 Techniques for Preventing Script Injection Vulnerabilities

Can we build web applications that are free from scripting vulnerabilities? Towards this goal, a majority of prior work on preventing scripting vulnerabilities has focused on fortifying the predominant defense deployed today called *sanitization*. Sanitization is the process of applying encoding or filtering primitives, called *sanitization primitives* or *sanitizers*, to render dangerous constructs in untrusted data inert [8, 129, 110, 125]. There are two well-established problems known about the practice of manually applying sanitizers which make it is notoriously prone to manual errors [8, 75, 62]. First, developers often implement sanitization primitives incorrectly [51, 8]. Second, developers often fail to apply *any* sanitization to untrusted content before embedding it inline in code that is parsed by the browser. We term this second problem as that of *missing sanitization* on application code paths. A significant body of prior research has focused on developing analysis for detecting program paths with missing sanitization [75, 62, 72, 132, 16, 85, 75, 121].

In this thesis, we identify new problems with sanitization based defenses. Specifically, we explain the subtleties of getting sanitization right using an abstract model of the web browser in Chapter 5. Our model is more precise than those used in previous works and it explains the issues with sanitization beyond those of just identifying program paths with missing sanitization. We also empirically analyze large-scale applications and emerging web application frameworks. We report on how often these subtle errors arise in our studied applications and application frameworks. In chapter 6, we propose a type based approach to automatically place sanitizers in application code, which is a principled step towards preventing the classes of errors we describing from arising. Our technique does not require any annotations to existing code and is designed to be bolted onto existing web frameworks such as Google Closure [44].

Prior and concurrent work has also investigated techniques to isolate trusted code from untrusted data in general, which relate to scripting defenses. These can be divided into three broad categories: server-side techniques [72, 110, 16], purely browser-based techniques [15, 79, 56] and client-server collaborative defenses [61, 49, 107]. We discuss the conceptual benefits and limitations of browser-only and server-only techniques in Section 7.7 of Chapter 7. We propose a conceptual framework for achieving a fundamental integrity property (called document structure integrity) in web applications via browser-server collaboration. This

techniques sidesteps the limitations of client- and server-only defenses. We combine and extend ideas from prior work on isolating inline untrusted content and confining it with security policies [61, 107, 105, 73]. We also build a proof-of-concept implementation that demonstrates the feasibility of such defense in practice.

Chapter 3

Finding Vulnerabilities using Taint-Enhanced Blackbox Fuzzing

In this chapter, we propose a *single-path analysis* technique which aims to generate an exploit input that traverses a given path in the program. Specifically, the input to a single-path analysis is an initial benign input that executes some path in the program. The analysis aims to find an exploit instance by systematically searching the equivalence class of inputs that forces program execution down the same path as the given benign input. We present this technique and our FLAX tool that implements it in this chapter.

We propose a dynamic analysis approach which we call *taint-enhanced blackbox fuzzing* for this problem. Our technique is a hybrid approach that combines the features of dynamic taint analysis [86, 132, 98, 118, 117] with those of automated random fuzzing [82]. It remedies the limitations of purely dynamic taint analysis, by using random fuzz testing to generate test cases that concretely demonstrate the presence of a client-side script injection vulnerability. This simple mechanism eliminates false alarms that would result from a purely taint-based tool.

The number of test cases generated by vanilla blackbox fuzzing increases combinatorially with the size of the input. In our hybrid approach, we use character-level precise dynamic taint information to prune the input search space significantly. Dynamic taint information extracts knowledge of the type of critical sink operation involved in the vulnerability, thereby making the subsequent blackbox fuzzing specialized for each sink type (or in other words, *sink-aware*). Taint-enhanced blackbox fuzzing scales because the results of dynamic taint analysis are used to create independent abstractions of the original application which are small and take fewer inputs, and can be tested efficiently with sink-aware fuzzing. From our experiments (Section 3.4), we see that the values to be fuzzed are on an average 55% smaller than the size of the original input. This reduction is achieved using character-level precise dynamic taint tracking.

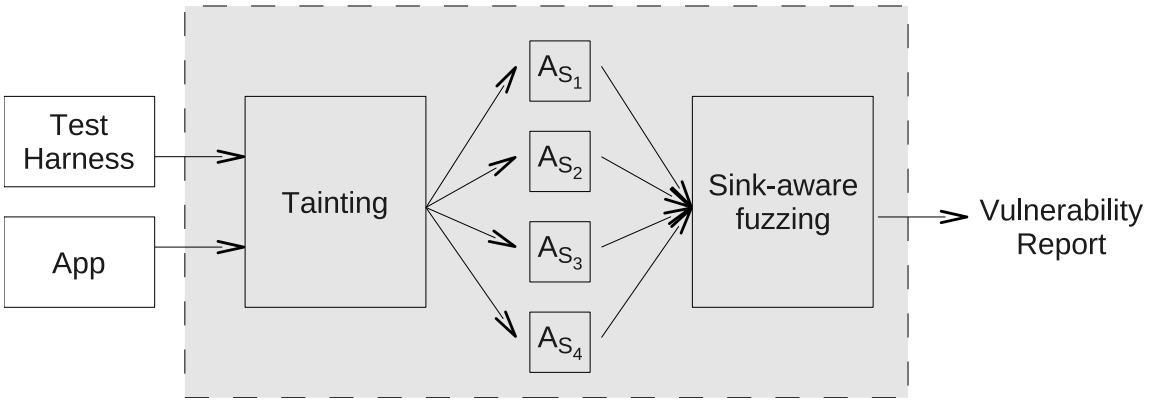

Figure 3.1: Approach Overview

3.1 Approach and Architectural Overview

Figure 3.1 gives a high-level view of our approach – the boxed, shaded part represents the primary technical contribution of this chapter. The input to our analysis is an initial benign input and the target application itself. The technique explores the equivalence class of inputs that execute the same program path as the initial benign input and finds a flow of untrusted data into a critical sink without sufficient validation.

Approach. In the first step, we execute the application with the initial input $\mathcal{I}$ and perform character-level dynamic taint analysis. Dynamic taint analysis identifies all uses of untrusted data in critical sinks[1]. This analysis identifies two pieces of information about each potentially dangerous data flow: the type of critical sink, and, the fractional part of the input that influences the data used in the critical sink. Specifically, we extract the range of input characters $\mathcal{I}_S$ on which data arguments of a sink operation $\mathcal{S}$ are directly dependent. All statements that operate on data that is directly dependent on $\mathcal{I}_S$, including path conditions involving conditional branches, are extracted into an executable slice of the original application which we term as an *acceptor slice* (denoted as $\mathcal{A}_S$). $\mathcal{A}_S$ is termed so because it is a stand-alone program that accepts inputs in the equivalence class of $\mathcal{I}$, in the sense that they execute the same program path as $\mathcal{I}$ up to the sink point $\mathcal{S}$. As the second step, we fuzz each $\mathcal{A}_S$ to find an input that exploits a bug. Our fuzzing is sink-aware because it uses the details of the sink node exposed by the taint analysis step. Fuzz testing on $\mathcal{A}_S$ semantically simulates fuzzing on the original application program. Using an acceptor slice to link the two high-level steps has two advantages:

- *Program size reduction.* $\mathcal{A}_S$ can be executed as a program on its own, but is significantly smaller in size than the original application. From our experiments in Section 4.6, $\mathcal{A}_S$ is typically smaller than the executed instruction sequence by a factor of 1000. Thus, fuzzing on a concise acceptor slice instead of the original complex application

[1]The definition of sinks in given in section 2.1

is a practical improvement. It avoids application restart, decouples the two high-level steps, and allows testing of multiple sinks to proceed in parallel.

- *Fuzzing search space reduction.* Sink-aware fuzzing focuses only on $\mathcal{I}_S$ for each $\mathcal{A}_S$, rather than the entire input. Additionally, our sink-aware fuzzer has custom rules for each type of critical sink because each sink results in different kinds of attacks and requires a different attack vector. As an example, it distinguishes `eval` sinks (which allow injection of JavaScript code) from DOM sinks (which allow HTML injection). Our sink-aware fuzzing employs input mutation strategies that are based on grammars such as the HTML syntax, JavaScript syntax, or URL syntax grammars.

3.2 Technical Challenges and Design Points

One of our contributions is to design a framework that simplifies JavaScript analysis and explicitly models reflected flows and path constraints. We explain each of these design points in detail below.

Modeling Path Constraints. The example defined in Figure 2.3 shows how validation checks manifest as conditional checks, affecting the choice of execution path in the program. Saner, an example of previous work that precisely analyzes server-side code, has considered only input-transformation functions as sanitization operations in its dynamic analysis, thereby ignoring branch conditions [8]. Our techniques improve on Saner's by explicitly modelling path constraints, thereby enabling Flax to capture the validation checks as branch conditions, as shown in the running example in the $\mathcal{A}_S$.

Simplifying JavaScript. There are two key problems in designing analyses for JavaScript code.

- *Rich data types and complex operations.* JavaScript supports complex data types such as string and array, with a variety of native operations on them. The ECMA-262 specification defines over 50 operations on string and array data types alone [34]. JavaScript analysis becomes complex because there are several syntactic constructs that can perform the same semantic operations. As a simple indicative example, there are several ways to split a string on a given separator (such as by using String.split, using String.match, and using String.indexOf with String.substring).

  In our approach, we canonicalize JavaScript operations and data references into a simplified intermediate form amenable for analysis, which we call *JASIL* (JAvascript Simplified Instruction Language). JASIL has a simpler type system and a smaller set of instructions which are sufficient to faithfully express the semantics of higher-level operations relevant to the applications we study. As a result, JASIL serves as a robust platform for simplified implementation of dynamic taint analysis and other analyses.

- *Aliasing.* There are numerous ways in which two different syntactic expressions can refer to the same object at runtime. This arises because of the dynamic features

of JavaScript, such as reflection, prototype-based inheritance, complex scoping rules, function overloading, as well as due to numerous exposed interfaces to access DOM elements. Reasoning about such a diverse set of syntactic variations is difficult. Previous static analysis techniques applied to this problem area required complex points-to analyses [47, 26]. In our dynamic analysis, we don't try to solve this static alias analysis problem; instead we record the concrete operand accessed during the execution of the program under the given benign input.

Avoiding alias analysis is an intentional choice in designing Flax. Flax dynamically translates JavaScript operations to JASIL, and by design each operand (an object, variable or data element) in JASIL is identified by its allocated storage address. With appropriate instrumentation of the JavaScript interpreter, we identify element accesses regardless of the syntactic complexity of the access pattern used in the references. For instance, details of whether a value lookup is executed by traversing the scope chain or the prototype chain is not recorded—only the storage memory address of the value is captured in JASIL.

Dealing with reflected flows. In this chapter, we consider data flows of two kinds: *direct* and *reflected*. A direct flow is one where there is a direct data dependency between a source operation and a critical sink operation in script code. Dynamic taint analysis identifies such flows as potentially dangerous. A reflected flow occurs when data is sent by the JavaScript application to a backend server for processing and the returned results are used in further computation on the client. Our dynamic taint analysis identifies untrusted data propagation across a reflected flow using a common-substring based content matching algorithm[2] [102]. During a reflected flow, data could be transformed on the server. The exact data transformation/sanitization on the server is hidden from the client-side analysis. To address this, we compositionally test the client-side code in two steps. First, we test the client-side code independently of the server-side code by generating candidate inputs that make simple assumptions about the transformations occurring in reflected flows. Subsequently, it verifies the assumption by running the candidate attack concretely, and reports a vulnerability if the concrete test succeeds.

3.3 Flax: Design and Implementation

We describe our algorithm for detecting vulnerabilities and present details about the implementation of our prototype tool Flax.

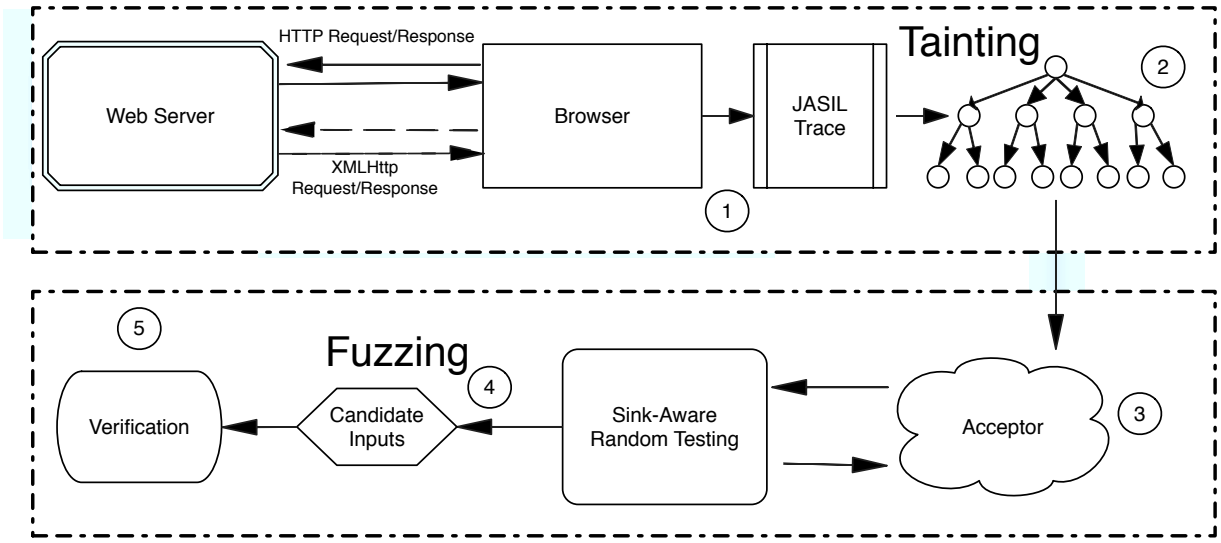

Figure 3.2: System Architecture for FLAX

Algorithm

Figure 3.2 shows the architectural overview of our taint-enhanced blackbox fuzzing algorithm. The pseudocode of the algorithm is described in Figure 3.3. At a high level, it consists of 5 steps:

1. *Dynamic trace generation and conversion to JASIL.* Run the application concretely in our instrumented web browser to record an execution trace in JASIL form.

2. *Dynamic taint analysis.* Perform dynamic taint analysis on the JASIL trace to identify uses of external data in critical sinks. For each such potentially dangerous data flow into a sink $\mathcal{S}$, our analysis computes the part of the untrusted input ($\mathcal{I}_\mathcal{S}$) which flows into the critical sink.

3. *Generate an acceptor slice.* For each sink $\mathcal{S}$ and the given associated information about $\mathcal{S}$ from the previous step, the analysis extracts an executable slice, $\mathcal{A}_S$, as defined in Section 3.1.

4. *Sink-aware random testing.* Apply random fuzzing to check if sufficient validation has been performed along the path to a given sink operation. For a given $\mathcal{A}_S$, our fuzzer generates random inputs according to sink-specific rules and custom attack vectors.

5. *Verification of candidate inputs.* Randomized testing of $\mathcal{A}_S$ generates candidate vulnerability inputs assuming a model of the transformation operations on the server that

[2]It is possible to combine client-side taint tracking with taint tracking on the server; however, in the present work we take a blackbox view of the web server.

```
Input: 𝒯 : Trace
Output: 𝒱 : AttackString List
type Flow : {
            var Sink, Source : Int List,
            var TaintedInsList : Int List,
            var InputOffset : (Int,Int) List
            };
var FlowList : Flow List;
FlowList = CalculateFlows (𝒯);
var Candidates = InputString List;
var 𝒱 = InputString List;
foreach ℱ in FlowList do
    𝒜_S = GenAutomaton(ℱ, 𝒯);
    Candidates = Fuzz ( 𝒜_S
    , max_length, max_iters);
    foreach 𝒞 in Candidates do
        𝒞_𝒯 = ExecuteOnInput(𝒞)
        var Result = VerifyAttack(𝒯, 𝒞_𝒯)
        if Result then
            𝒱.append([ℱ, 𝒞_𝒯.input]);
        end
    end
end
return 𝒱;
```

Figure 3.3: Algorithm for FLAX

x : τ ::=	v : τ	(Assignment, Type Conversion)
x : τ ::=	$*$ (v : $Ref(\tau)$)	(Dereference)
x : *Int* ::=	v1 : *Int* op v1 : *Int*	(Arithmetic)
x : *Bool* ::=	v1 : τ op v1 : τ	(Relational)
x : *Bool* ::=	v1 : *Bool* op v1 : *Bool*	(Logical)
x : PC ::=	if (testvar : *Bool*)	
	then (c : *Int*) else (c : *Int*)	(Control Flow)
x : *String* ::=	substring(s : *String*,	
	startpos : *Int*, len : *Int*)	(String Ops)
x : *String* ::=	concat(s1 : *String*, s2 : *String*)	(String Ops)
x : *String* ::=	fromArray(s1 : $Ref(\tau)$)	(String Ops)
x : *String* ::=	convert(s1 : *String*)	(String Ops)
x : *Char* $*$ κ ::=	convert(i : *Int*)	(Character Ops)
x : *Int* ::=	convert(i : *Char* $*$ κ)	(Character Ops)
x : τ ::=	$\mathcal{F}$ (i_1 : τ, ..., i_n : τ)	(Uninterpreted Function Call)

Figure 3.4: Simplified operations supported in JASIL intermediate representation

$$
\begin{array}{|lll|}
\hline
\tau & := & \eta \mid \beta[\eta] \mid Bool \mid Null \mid Undef \mid PC \\
\eta & := & Int \mid \beta \\
\beta & := & Ref(\tau) \mid String(\kappa) \mid Char(\kappa) \\
\kappa & := & UTF8 \mid UTF7 \mid \ldots \\
\hline
\end{array}
$$

Figure 3.5: Type system of JASIL intermediate representation

may occur in reflected flow. This final step verifies that the assumptions hold, by testing the attacks concretely on the web application and checking that the attack succeeds by using a browser-based oracle.

JASIL

To simplify further analysis, we lower the semantics of the JavaScript language to a simplified intermediate representation which we call JASIL. JASIL is designed to have a simple type system with a minimal number of operations on the defined data types. A brief summary of its type system and categories of operations are outlined in Figure 4.7 and Figure 3.4 respectively. JavaScript interpreters already perform some amount of semantic lowering in converting to internal bytecode. However, the semantics of typical JavaScript bytecode are not substantially simpler, because most of the complexity is hidden in the implementation of the rich native operations that the interpreter's runtime supports.

JASIL has a substantially smaller set of operations, shown in Figure 3.4. In our design, we have found JASIL to be sufficient to express the operational semantics of a subset of JavaScript commonly used in real applications. Our design is implemented using WebKit's JavaScript interpreter, the core of the Safari web browser, and is faithful to the semantics of the operations as implemented therein. In our work, we lower all the native string operations, array operations, integer operations, regular expression based operations, global object functions, DOM functions, and the operations on native window objects. Lowering to JASIL simplifies analyses. For instance, consider a `String.replace` operation in JavaScript. A replace operation retains some parts of its input string in its output while transforming the other parts with specified strings. An execution of the replace operation is represented in JASIL as a series of substring operations on the inputs followed by a final concatenation of these intermediate substrings. For extracting the start and end indices of these intermediate substrings, we have instrumented the implementation of the replace function in the JavaScript interpreter after it performs the matching operation on the input[3]. With JASIL, subsequent dynamic taint analysis is simplified because the tainting engine only needs to reason about simple operations like substring extraction and concatenation rather than the semantics of the matching algorithm in replace. An alternative to such simplification is to model the replace operation as a transducer [8].

[3]The matching operation may call a regular expression engine; we record the start and end indices after the regular expression engine runs over the given input.

Sources	**Critical Flow Sinks**	**Resulting Exploit**
document.URL document.URLUnencoded document.location.* document.referrer.* window.location.* event.data event.origin textbox.value forms.value	eval(), window.execScript(), window.setInterval(), window.setTimeout()	Script Injection
	document.write(...), document.writeln(...), document.body.innerHtml, document.cookie document.forms[0].action, document.create(), document.execCommand(), document.body.*, window.attachEvent(), document.attachEvent()	HTML code Injection
	document.cookie	Session fixation
	XMLHttpRequest.open(,url,), document.forms[*].action,	HTTP Param Injection

Figure 3.6: (Left) Sources of untrusted data. (Right) Critical sinks and corresponding exploits that may result if untrusted data is used without proper validation.

In addition to lowering semantics of complex operations, JASIL explicitly models procedure call/return semantics, parameter evaluation, parameter passing, and object creation and destruction. Property look-ups on JavaScript objects and accesses to native objects such as the DOM or window objects are converted to operations on a functional map in JASIL (denoted by $\beta[\eta]$ in its type system). This canonicalization of references makes further analysis easier.

In JASIL, each object, variable or data element is identified by its allocated storage address, which obviates the need to reason about most forms of aliasing. As one example of how this simplification allows robust reasoning, consider the case of prototype-based inheritance in JavaScript. In JavaScript, whenever an object $\mathcal{O}$ is created, the object inherits all the properties of a *prototype object* corresponding to the constructor function, accessible through the `.prototype` property of the function (functions are first-class types in JavaScript and behave like normal objects). The prototype object of the constructor function could in turn inherit from other prototype objects depending on how they are created. When a reference $\mathcal{O}$.`f` is resolved, the field `f` is first looked up in the object $\mathcal{O}$. If it is not found, it is looked up in the prototype object of $\mathcal{O}$ and in the subsequent objects of the prototype chain. Thus, determining which object is referenced by $\mathcal{O}$ statically requires a complex alias analysis. In simplifying to JASIL, we instrumented the interpreter to record the address identifier for each variable used after the reference resolution process (including the scope and prototype chain traversals) is completed. Therefore, further analysis does not need any further reasoning about prototypes or scopes.

To collect a JASIL trace of a web application for analysis we instrumented the browser's JavaScript interpreter to translate the bytecode executed at runtime to JASIL. This required instrumentation of the JavaScript interpreter, bytecode compiler and runtime, resulting in

a patch of 6032 lines of C++ code to the vanilla WebKit browser. To facilitate recovering JavaScript source form from the JASIL representation, auxiliary information mapping the dynamic allocation addresses to native object types is embedded as metadata in the JASIL trace.

Dynamic taint analysis

Character-level precise modeling of string operation semantics. JavaScript applications are array- and string- centric; lowering of JavaScript to JASIL is a key factor in reasoning about complex string operations in our target applications. Dynamic taint analysis has been used with success in several security applications outside of the realm of JavaScript applications [132, 87, 86]. For JavaScript, Vogt *et al.* have previously developed taint-tracking techniques to detect confidentiality attacks resulting from cross-site scripting vulnerabilities[118]. In contrast to their work, our techniques model the semantics of string operations and are character-level precise.

We list the taint sources and sinks used by default in Flax in Figure 3.6. Flax models only direct data dependencies for this step; additional control dependencies for path conditions are introduced during $\mathcal{A}_S$ construction. It performs taint-tracking offline on the JASIL execution trace, which reduces the intrusiveness of the instrumentation by not requiring transformation of the interpreter's core semantics to support taint-tracking. Taint propagation rules are straight-forward—assignment and arithmetic operations taint the destination operand if one of the input operands is tainted, while preserving character-level precision. The JASIL string `concatenation` and `substring` operations result in a merge and slicing operation over the ranges of tainted data in the input operands, respectively. The `convert` operation, which implements character-to-integer and integer-to-character conversion, typically results from simplifying JavaScript encode/decode operations (such as `decodeURI`). Taint propagation rules for `convert` are similar: the output is tainted if the input is tainted. Other native functions that are not explicitly modeled are treated as uninterpreted transfer functions, acting merely to transfer taint from input parameters to output parameters in a conservative way. Recall that operations such as regular-expression based match and replace are already lowered to substrings and concatenations in JASIL by tracing the JavaScript interpreter's execution.

Tracking data in reflected flow. During this analysis data may be sent to a backend server via the `XMLHttpRequest` object. We approximate taint propagation across such network data flows by using an exact substring match algorithm, which is a simplified form of black-box taint inference techniques proposed in the previous literature [106, 102]. We record all tainted data sent in a reflected flow, and perform a longest common substring match on the data returned. Any matches that are above a threshold length are marked as tainted, and the associated taint metadata is propagated to the reflected data. This technique has proved sufficient for the AJAX applications in our experiments.

Implicit Sinks. Certain source operations do not have explicit sink operations. For in-

```
function acceptor (input) {
  var path_constraints = true;
  var re = /(.*?):\/\/(.*?)\.com/;
  var matched = re.exec(input);
  if (matched == null) {
  path_constraints = path_constraints & false;
  }
  if (!path_constraints) return false;
  var domain = matched[2];
  var valid = /example/.test(domain);
  path_constraints = path_constraints & valid;
  if (!path_constraints) return false;
  var port = matched[1];
  valid = /https?/.test(port);
  path_constraints = path_constraints & valid;
  if (!path_constraints) return false;
  return true;
}
```

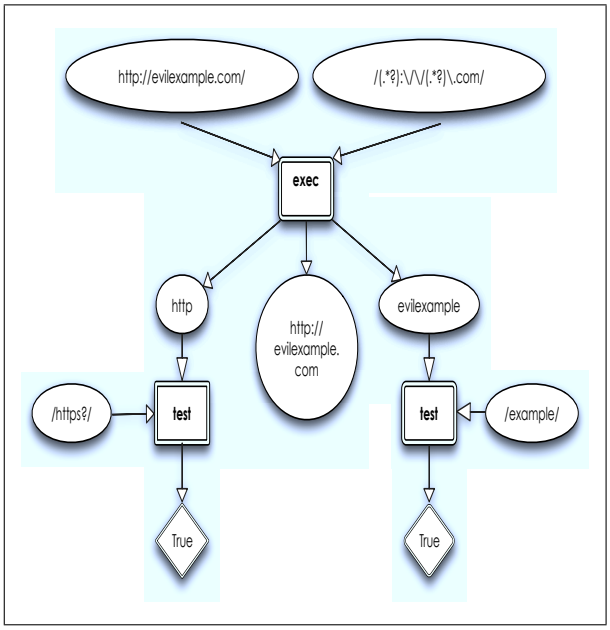

Figure 3.7: (Left) Acceptor Slice showing validation and parsing operations on `event.origin` field in the running example. (Right) Execution of the Acceptor Slice on a candidate attack input, namely `http://evilexample.com/`

stance, in the example of Figure 2.3 the `event.origin` field has no explicit sink. However, this field must be sanitized before any use of `event.data`. We model this case of implicit dependence between two fields by introducing an *implicit sink* node for `event.origin` at any use of `event.data` in critical sink operation. This has the effect that for any use of `event.data`, the path constraint checks on `event.origin` are implicitly included in the acceptor slice.

Acceptor Slice Construction

After dynamic taint analysis identifies a sink point, FLAX extracts a dynamic executable slice from the program, by walking backwards from the critical sink to the source of untrusted data. In order to fuzz the slice, the JASIL slice is converted back to a stand-alone JavaScript function. This results in an executable function that retains the operations on $\mathcal{I}_\mathcal{S}$, and returns `true` for many (not all) inputs that execute the same path as the original run. The slicing operation captures (a) *data dependencies*, i.e., all operations directly processing $\mathcal{I}_\mathcal{S}$ and (b) a limited form of control dependencies, i.e., all path constraints, conditions of which are directly data dependent on $\mathcal{I}_\mathcal{S}$ that affect control-flow decisions. Path constraints are conditional checks corresponding to each branch point (including indirect function calls) which force the execution to take the same path as $\mathcal{I}_\mathcal{S}$. Data values which are not directly data dependent (marked tainted) in the original execution, are replaced with their concrete

constant values observed during the program execution.

Acceptor Slice for the Running Example. The instructions operating on the value of `event.origin` in the running example that influences the implicit `eval` sink is shown in Figure 3.7. It shows the $\mathcal{A}_S$ for the `event.origin` field of our example, after certain optimizations, like dead-code elimination. This program models all the validation checks performed on that field, until its use in the implicit sink node at `eval`.

Sink-aware fuzzing

This step in our analysis performs randomized testing on each $\mathcal{A}_S$. Note that each critical sink operation can result in a different kind of vulnerability. Therefore, it is useful to target each sink node ($\mathcal{S}$) with a set of specialized attack vectors. For instance, an unchecked flow that writes to the `innerHTML` property of a DOM element can result in HTML code injection and our fuzzer attempts to inject an HTML tag into such a sink. For `eval` sink, our testing targets the injection of JavaScript code. We incorporate a large corpus of publicly available attack vectors for XSS [94] in our fuzzing.

While testing for an attack input that causes $\mathcal{A}_S$ to return true, our fuzzer utilizes the aforementioned attack vectors and a grammar-aware strategy. Starting with the initial benign input, the fuzzer employs a mutation-based strategy to transform, prepend and appends language nonterminals. For each choice, the fuzzer first selects terminal characters based on the knowledge of surrounding text (such as HTML tags, JavaScript nonterminals) and finally resorts to random characters if the grammar-aware strategy fails to find a vulnerability.

To check if a candidate attack input succeeds we use a browser-based oracle. Each candidate input is executed in $\mathcal{A}_S$ and the test oracle determines if the specific attack vector is evaluated or not. If executed, the attack is verified as being a concrete attack instance. For instance, in our running example, the `event.origin` acceptor slice returns true for any URL principal which is not a subdomain of `http://example.com`[4]. Our fuzzer tries string mutations of the original domain `http://example.com` and quickly discovers that there are other domains that circumvent the validation checks.

3.4 Evaluation

Our primary objective is to determine if taint-enhanced blackbox fuzzing is scalable enough to be used on real-world applications to discover vulnerabilities. As a second objective, we aim to quantitatively measure the benefits of taint-enhanced blackbox fuzzing over vanilla taint-tracking and purely random testing. In our experiments, Flax discovers 11 previously unknown vulnerabilities in real applications and our results show that our design of taint-enhanced blackbox fuzzing offers significant practical gains over vanilla taint-tracking and

[4]Recall that the running example acceptor does not have an explicit sink, therefore it only returns `true` on success and false otherwise.

Name	# of Taint Sinks	**Verified Vuln.**	Size of Total Inputs	Size of Acceptor
Plaxo	178	**0**	119	60
Academia	1	**1**	334	21
Facebook Chat	44	**0**	127	127
ParseURI	1	**1**	78	62
AjaxIM	20	**2**	28	28
AskAWord	3	**1**	26	26
Block Notes	1	**1**	474	96
Birthday Reminder	6	**0**	632	246
Calorie Watcher	3	**0**	681	20
Expenses Manager	6	**0**	1,137	65
MyListy	1	**1**	578	47
Notes LP	5	**0**	740	30
Progress Bar	151	**0**	496	264
Simple Calculator	1	**1**	27	27
Todo List	1	**0**	632	40
TVGuide	2	**1**	586	66
Word Monkey	1	**1**	26	26
Zip Code Gas	5	**1**	412	69

Name	Trace Size (# of insns)	Avg. size of $\mathcal{A}_S$ Inputs	# of Tests to Find 1st Vulnerability	Vulnerability Type
Plaxo	557,442	36	-	-
Academia	156,621	286	16	Origin Mis-attribution
Facebook Chat	6,460,591	1,151	-	-
ParseURI	55,179	638	6	Code injection
AjaxIM	223,504	517	93	Code injection , Application Command Injection
AskAWord	59,480	611	93	Cookie Sink
Block Notes	11,539	766	28	Code injection
Birthday Reminder	2,178,927	664	-	-
Calorie Watcher	449,214	733	-	-
Expenses Manager	522,788	1,454	-	-
MyListy	17,054	1,468	4	Code injection
Notes LP	144,829	3,327	-	-
Progress Bar	118,108	475	-	-
Simple Calculator	72,475	4	93	Code injection
Todo List	647,849	1,181	-	-
TVGuide	24,144,843	188	8,366	Code injection
Word Monkey	237,837	99	93	Code injection
Zip Code Gas	410,951	248	2	Code injection

Table 3.1: Applications for which FLAX observed untrusted data flow into critical sinks. The top 5 subject applications are websites and the rest are iGoogle gadgets. (Upper) Columns 2-5, and (Lower) Columns 6-9.

fuzzing. We also investigate the security implications of the vulnerabilities by constructing proof-of-concept exploits and we discuss their varying severity in this section.

Test Subjects

We selected a set of 40 web applications consisting of iGoogle gadgets and other AJAX applications for our experiments. Of these, FLAX observed untrusted data flows into critical sinks for only 18 of the cases, consisting of 13 iGoogle gadgets and 5 web applications. We report detailed results for only these 18 applications in Table 3.1. We tested each subject application manually to explore its functionality, giving benign inputs to seed our automated testing. For instance, all of the iGoogle gadgets were tested by visiting the benign URL used by the iGoogle web page to embed the gadget in its page. To explore each application's functionality, we manually entered data into text boxes, clicked buttons and hyperlinks, simulating the behavior of a normal user. These manual test cases served as the initial benign test inputs for our analysis.

Google gadgets constitute the largest fraction of our study because they are simple applications which are popular among internet users today. Most gadgets are reported to have thousands of users with one of the vulnerable gadgets having over 1,350,000 users, as per the data available from the iGoogle gadget directory on December 17^{th} 2009 [57]. The other AJAX applications consist of social networking sites, chat applications and utility libraries which are examples of the trend towards increasing code sharing via third-party libraries. All tests were performed using our FLAX framework running on a Ubuntu 8.04 platform with a 2.2 GHz, 32-bit Intel dual-core processor and 2 GB of RAM.

Experimental Results

FLAX found several distinct taint sinks in the applications, only a small fraction of which are deemed vulnerable by the tool. Column 2 and 3 of Table 3.1 reports the number of distinct sinks and number of vulnerabilities found by FLAX respectively. The use of character-level precise taint tracking in FLAX prunes a significant fraction of the input in several cases for further testing. To quantitatively measure this saving we observe the average sizes of the original input and the reduced input size in the acceptor slices (used for subsequent fuzzing), which is reported in columns 4 and 5 of Table 3.1 respectively. We measure the reduction in the acceptor size, which results in substantial practical efficiencies in subsequent black-box fuzzing. We find that the acceptor slices are small enough to often enable manual analysis for a human analyst. Columns 6 and 7 report the size of dynamic execution trace and the average size of the acceptor slices respectively.[5] The last two columns in Table 3.1 show the number of test cases it takes to find the first vulnerability in each application and the kinds of vulnerability found.

[5]In our implementation, the acceptor slices are converted back to JavaScript form for further analysis: the size of acceptor slices increases as a result of this conversion by a factor of 4 at most in our implementation, as compared to the numbers reported in column 7

Prevalence of client-side script injection vulnerabilities

Of the 18 applications in which FLAX observed a dangerous flow, it found a total of 11 vulnerabilities which we report in the third column of Table 3.1. The vulnerabilities are evidence of a broad range of attack possibilities, though code injection vulnerabilities were the highest majority. FLAX reported 8 code injection vulnerabilities, 1 origin mis-attribution vulnerability, 1 cookie-sink vulnerability and 1 application command injection vulnerability. We confirmed that all vulnerabilities reported were true positives by manually inspecting the JavaScript code and concretely evaluating them with exploit inputs. The severity of the vulnerabilities varied by application and source of untrusted input, which we discuss in section 3.4.

Effectiveness

We quantitatively measure the benefits of taint-enhanced blackbox fuzzing over vanilla taint-tracking and random fuzzing from our experimental results.

False Positives Comparison. The second column in Table 3.1 shows the number of distinct flows of untrusted data into critical sink operations observed; only a fraction of these are true positives. Each of these distinct flows is an instance where a conservative taint-based tool would report a vulnerability. In contrast, the subsequent step of sink-aware fuzzing in FLAX eliminates the spurious alarms, and a vulnerability is reported (column 3 of Table 3.1) only when a witness input is found. It should be noted that FLAX can have false negatives and could have missed bugs, but completeness is not an objective for FLAX.

We manually analyzed the taint sinks reported as safe by FLAX and, to the best of our ability, found them to be true negatives. For instance, we determined that most of the sinks reported for the Plaxo case were due to code which output the length of the untrusted input to the DOM, which executed repeatedly each time the user typed a character in the text box. Many of the true negatives we manually analyzed employed sufficient validation – for instance, Facebook Chat application correctly validates the origin property of every `postMessage` event it received in the execution. Several other applications validate the structure of the input before using it in a JavaScript `eval` statement or strip dangerous characters before using it in HTML code evaluation sinks.

Efficiency of sink-aware fuzzing. Table 3.1 (column 8) shows the number of test cases FLAX generated before it found the vulnerability for the cases it deems unsafe. Part of the reason for the small number of cases on average, is that our fuzzing leverages knowledge of the sink operations. Column 4 of the Table 3.1 shows that the size of the original inputs for most applications is in the range of 100-1000 characters. Slicing on the tainted data prunes away a significant portion of the input space, as seen from column 5 of Table 3.1. We report an average reduction of 55% from the original input size to the size of test input used in acceptor slices.

Further, the average size of an acceptor slice (reported in column 7 of Table 3.1) is smaller than the original execution trace by approximately 3 orders of magnitude. These

```
function acceptor(input) {
 //input = '{"action":"","val":""}';
 must_match = '{]:],]:]}';
 re1 =/\\(?:["\\\/bfnrt]|u[0-9a-fA-F]{4})/g;
 re2 =/"[^"\\\n\r]*"|true|false|null|
      -?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?/g;
 re3 = /(?:^|:|,)(?:\s*\[)+/g;
 rep1 = input.replace(re1, "@");
 rep2 = rep1.replace(re2, "]");
 rep3 = rep2.replace(re3,"");
 if(rep3 == must_match) { return true; }
 return false;
}
```

Figure 3.8: An example of a acceptor slice which uses complex string operations for input validation, which is not directly expressible to the off-the-shelf string decision procedures available today.

reductions in test program size for sink-aware fuzzing allow sink-aware fuzzing to work with much smaller abstractions of the original application, thereby significantly improving the efficiency of this step.

Qualitative comparison to other approaches. Figure 3.8 shows one of the several examples that Flax generates which can not be directly expressed to the languages supported by off-the-shelf existing string decision procedures [66, 50], which Flax deems as safe. We believe that even human analysis for such cases is tedious and error-prone.

Security Implication Evaluation and Examples

To gain insight into their severity we further analyzed the vulnerabilities reported by Flax and created proof-of-concept exploits for a few of them to validate the threat. All vulnerabilities were disclosed to the developers either through direct communication or through CERT.

Origin Mis-attribution in Facebook Connect. Flax reported an origin mis-attribution vulnerability for *academia.edu*, a popular academic collaboration and document sharing web site used by several academic universities. Flax reported that the application was vulnerable due to a missing validation check on the `origin` property of a received postMessage event. We manually created a proof-of-concept exploit which demonstrates that any remote attacker could inject arbitrary script code into the vulnerable web application. On further analysis, we found that the vulnerability existed in the code for Facebook Connect library, which was used by academia.edu as well as several other web applications. We disclosed the vulnerability to Facebook developers on December 15^{th} 2009 and they released a patch for the vulnerability within 6 hours of the disclosure.

Script Injection. FLAX reported 8 code injection vulnerabilities (DOM-based XSS) in our target applications, where untrusted values were written to code evaluation constructs in JavaScript (such as `eval`, `innerHTML`). One DOM-based XSS vulnerability was found on each of the following: 6 distinct iGoogle gadgets, an AJAX chat application (AjaxIM), and one URL parsing library's demonstration page. We manually verified that all of these were true positives and resulted in script execution in the context of the vulnerable domains, when the untrusted source was set with a malicious value. Four of the code injection vulnerabilities were exploitable when remote attackers entice the user into clicking a link of an attacker's choice. The affected web applications were also available as iGoogle gadgets and we discuss a *gadget overwriting* attack using client-side script injection vulnerabilities below. The remaining 4 code injection vulnerabilities were self-XSS vulnerabilities as the untrusted input source was user-input from a form field, a text box, or a text area. As explained in section 2.1, these vulnerabilities do not directly empower a remote attacker without additional social engineering (such as enticing users into copy-and-pasting text). All gadget developers we were directly able to communicate with positively acknowledged the concern and agreed to patch the vulnerabilities.

Gadget Overwriting Attacks. In a gadget overwriting attack, a remote attacker compromises a gadget and replaces it with the content of its choice. We assume the attacker is an entity which controls a web-site and has the ability to entice the victim user into clicking a malicious link. We describe a gadget overwriting attack with an example of how it can be used to create a phishing attack layered on the gadget's client-side script injection vulnerability. In a gadget overwriting attack, the victim clicks an untrusted link, just as in a reflected XSS attack, and sees a page such as the one shown in Figure 3.9 in his browser. The URL bar of the page points to the legitimate iGoogle web site, but the gadget has been compromised and displays attacker's contents: in this example, a phishing login box which tempts the user to give away his credentials for Google. If the user enters his credentials, they are sent to the attacker rather than Google or the gadget's web site. The attack mechanics are as follows. First, the victim visits the attacker's link which points to the vulnerable gadget domain (typically hosted at a subdomain of *gmodules.com*). The link exploits a code injection client-side script injection vulnerability in the gadget and the attack payload is executed in the context of the gadget's domain. The attacker's payload then spawns a new window which points to the full iGoogle web page (`http://www.google.com/ig`) containing several gadgets including the vulnerable gadget in separate `iframes`. Lastly, the attacker's payload replaces the content of the vulnerable gadget's iframe in the new window with contents of its choice. This cross-window scripting is permitted by browser's same-origin policy because the attacker's payload and the gadget's iframe principal are the same.

We point out that Google/IG is designed such that each iGoogle gadget runs as a separate security principal hosted at a subdomain of *http://gmodules.com*. This mitigation prevents an attacker who compromises a gadget from having any access to the sensitive data of the *google.com* domain. In the past, Barth *et al.* described a related attack, called a *gadget*

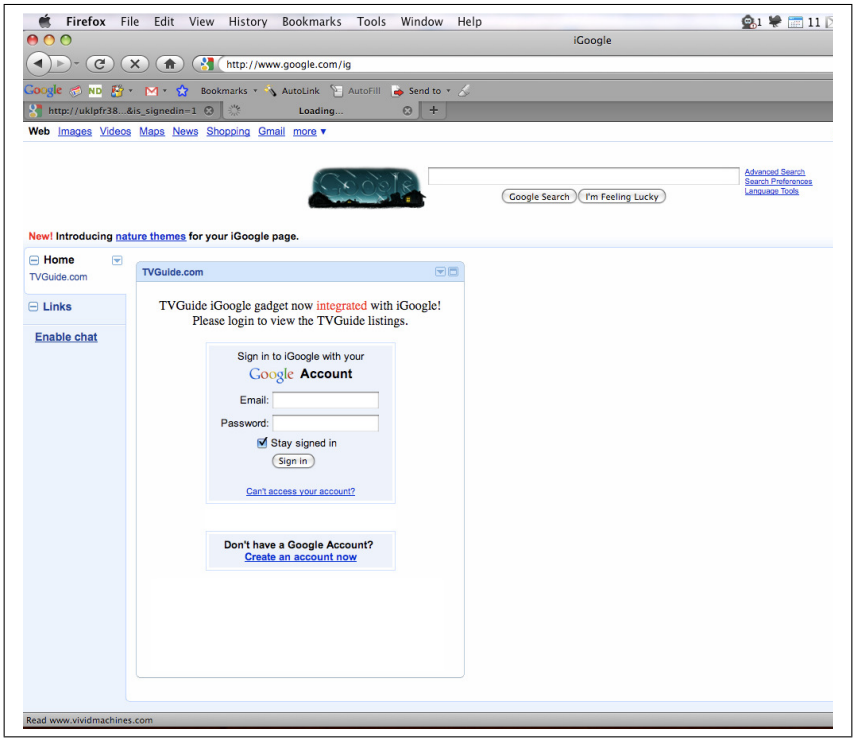

Figure 3.9: A gadget overwriting attack layered on a client-side script injection vulnerability. The user clicks on an untrusted link which shows the iGoogle web page with an overwritten iGoogle gadget. The URL bar continues to point to the iGoogle web page.

hijacking attack, which allows attackers[6] to steal sensitive data by navigating the gadget frame to a malicious site [13]. Barth *et al.* proposed new browser frame navigation policies to prevent these attacks. Gadget overwriting attacks resulting from client-side script injection vulnerabilities in vulnerable gadgets can also allow attackers to achieve the same attack objectives as those remedied by the defenses proposed by Barth *et al.* [13].

Cookie-sink Vulnerabilities. FLAX reported a cookie corruption vulnerability in one of AskAWord iGoogle gadgets which provide the AskAWord.com dictionary and spell checker service. FLAX reported that the cookie data could be corrupted with arbitrary data and additional cookie attributes could be injected, which is a low severity vulnerability. However, on further analysis, we found that the gadget used the cookie to store the user's history of previous searches which was echoed back on the server's HTML response without any client-side or server-side validation. We subsequently informed the developers about the cookie attribute injection and the reflected XSS vulnerability through the cookie channel, and the developers patched the vulnerability on the same day.

[6]A gadget attacker described by Barth *et al.* requires the privilege that the integrator embeds a gadget of the attacker choice, which is different from the attacker model in a gadget overwriting attack

HTTP Parameter Pollution. One vulnerability reported by Flax for AjaxIM chat application indicated that such bugs can result in practice. Flax reported that untrusted data from an input text box could be used to inject application commands. AjaxIM uses untrusted data to construct a URL that directs application-specific commands to its backend server using `XMLHttpRequest`. These commands include adding/deleting chat rooms, adding/deleting friends and changing the user's profiles. Flax discovered a vulnerability where an unsanitized input from an input-box is used to construct the URL that sends a GET request command to join a chat room. An attacker can exploit this vulnerability by injecting new HTTP parameters (key-value pairs) to the URL. A benign command request URL to join a chat room named `'friends'` in AjaxIM is of the form `ajaxim.php?call=joinroom&room=friends`. We confirmed that by providing a room name as `'friends&call=addbuddy&buddy=evil'` results in overriding the value of the `call` command from 'joinroom' to a command that adds an untrusted user (called "evil") to the victim's friend list.

The severity of this vulnerability is very limited as it does not allow a remote attacker to exploit the bug without additional social engineering. However, we informed the developers and they acknowledged the concern agreeing to fix the vulnerability.

3.5 Related Work

Client-side script injection vulnerabilities constitute attack categories that have similar counterparts in server-side application logic — this has driven a majority of the research on web vulnerabilities to analysis of server-side logic written in languages such as PHP. First, we discuss the techniques employed in these and compare it our taint-enhanced blackbox fuzzing. Next, we compare the benefits of our approach with purely taint-based analysis approaches, and other semi-random testing based approaches. Finally, we discuss the recent frameworks proposed for analysis of JavaScript applications.

Server-side vulnerabilities. XSS, SQL injection, directory traversal, cross-site request forgery and command injection have been the most important kind of web vulnerabilities in the last few years[108]. Techniques including static analyses [62, 54], model checking [75], mixed static-dynamic analyses [8], as well as decision procedure based automated analyses [66, 50] have been developed for server-side applications written in PHP and Java. Of these techniques, only a few works have aimed to precisely analyze custom validation routines. Balzarotti *et al.* were the first to identify that the use of custom sanitization could be an important source of both false positives and negatives for analysis tools in their work on Saner[8]. The proposed approach used static techniques for reasoning about multiple paths effectively. However, the sanitization analysis was limited to a subset of string functions and ignored validation checks that manifest as conditional constraints on the execution path. Though an area of active research, the more recent string decision procedures do not yet support the full generality of constraints we practically observed in our JavaScript subject applications [66, 50, 17].

Dynamic taint analysis approaches. Vogt *et al.* have developed taint-analysis techniques for JavaScript to study the problem of confidentiality attacks resulting from XSS vulnerabilities [118]. In addition to the features provided by their work, our taint-tracking techniques are character-level precise and accurately model the semantics of string operations as our application domain requires such precision. Purely dynamic taint-based approaches have been used for runtime defense against web attacks [132, 87, 117, 110, 49, 84, 107]. However, applying these to discover attacks is difficult because reasoning about validation checks is important for precision. Certain tools such as PHPTaint [117] approximate this by implicitly clearing the taint when data is sanitized using a built-in sanitization routine.

JavaScript analysis frameworks. Several works have recently applied static analysis on JavaScript applications [47, 26]. In contrast, we demonstrate the practical effectiveness of a complimentary dynamic analysis technique and we explain the benefits of our analyses over their static counterparts. GateKeeper enforces a different set of policies using static techniques which may lead to false positives. Recent frameworks for dynamic analyses [134] have been proposed for source-level instrumentation for JavaScript; however, source-level transformations are much harder to reason about in practice due to the complexity of the JavaScript language.

Browser vulnerabilities. Client-side script injection vulnerabilities are related to, but significantly different from browser vulnerabilities [13, 120, 25, 11]. Research on these vulnerabilities has largely focused on better designs of interfaces that could be used securely by mutually untrusted principals. In this chapter, we showed how web application developers use these abstractions, such as inter-frame communication interfaces, in an insecure way.

3.6 Conclusion

This chapter presents a new hybrid approach to automatically test JavaScript applications for the presence of client-side script injection vulnerabilities. We implemented our approach in a prototype tool called Flax. Flax has discovered several real-world bugs in the wild, which suggests that such tools are valuable resources for security analysts and developers of rich web applications today. Results from running Flax provide key insight into the prevalence of this class of client-side script injection vulnerabilities with empirical examples, and point out several implicit assumptions and programming errors that JavaScript developers today make.

Chapter 4

Finding Vulnerabilities using Dynamic Symbolic Execution

In the previous chapter, we propose a technique that analyzes a single-path of execution in the application to detect a client-side script injection vulnerability. The proposed techniques assumes that an external test harness is available externally to explore the execution space of the application. In this chapter, we present techniques and a system for automatically exploring the execution space of client-side JavaScript code. The technique proposed in the chapter, though more heavy-weight, automatically generates a test harness from an initial benign input to automatically explore the execution space of the program. Our focus, as with the previous chapter, is on JavaScript applications and we demonstrate its effectiveness in finding client-side script injection vulnerabilities.

JavaScript execution space exploration is challenging for many reasons. In particular, JavaScript applications accept many kinds of input, and those inputs are structured just as strings. For instance, a typical application might take user input from form fields, messages from its server via `XMLHttpRequest`, and data from code running concurrently in other browser windows. Each kind of input string has its own format, so developers use a combination of custom routines and third-party libraries to parse and validate the inputs they receive. To effectively explore a program's execution space, a tool must be able to supply values for all of these different kinds of inputs and reason about how they are parsed and validated.

Approach. In this chapter, we develop a dynamic symbolic-execution based framework for client-side JavaScript code analysis. We build an automated, stand-alone tool called Kudzu that, given a URL for a web application, automatically generates high-coverage test cases to systematically explore its execution space. Automatically reasoning about the operations we see in real JavaScript applications requires a powerful constraint solver, especially for the theory of strings. However, the power needed to express the semantics of JavaScript operations is beyond what existing string constraint solvers [66, 50] offer. As a central contribution of this work, we overcome this difficulty by proposing a constraint

language and building a practical solver (called KALUZA) that supports the specification of boolean, machine integer (bit-vector), and string constraints, including regular expressions, over multiple variable-length string inputs. This language's rich support for string operations is crucial for reasoning about the parsing and validation checks that JavaScript applications perform.

To show the practicality of our constraint language, we detail a translation from the most commonly used JavaScript string operations to our constraints. This translation also harnesses concrete information from dynamic execution traces of the program in a way that allows the analysis to scale. We analyze the theoretical expressiveness of the theory of strings supported by our language (including in comparison to existing constraint solvers), and bound its computational complexity. We then give a sound and complete decision procedure for the bounded-length version of the constraint language. We develop an end-to-end system, called KUDZU, that performs symbolic execution with this constraint solver at its core.

End-to-end system. We identify further challenges in building an end-to-end automated tool for rich web applications. For instance, because JavaScript code interacts closely with a user interface, its input space can be divided into two classes, the *events space* and the *value space*. The former includes the state (check boxes, list selections) and sequence of actions of user-interface elements, while the latter includes the contents of external inputs. These kinds of input jointly determine the code's behavior, but they are suited to different exploration techniques. KUDZU uses GUI exploration to explore the event space, and symbolic execution to explore the value space.

We evaluate KUDZU's end-to-end effectiveness by applying it to the collection of 18 JavaScript applications we studied in Section 3.4. The results show that KUDZU is effective at getting good coverage by discovering new execution paths, and it automatically discovers 2 previously-unknown vulnerabilities, as well as 9 client-side script injection vulnerabilities that were previously found only with manually-created test-cases running under FLAX.

In summary, this chapter makes the following main contributions:

- We identify the limitations of previous string constraint languages that make them insufficient for parsing-heavy JavaScript code, and design a new constraint language to resolve those limitations. (Section 4.3)
- We design and implement KALUZA, a practical decision procedure for this constraint language. (Section 4.4)
- We build the first symbolic execution engine for JavaScript, using our constraint solver. (Sections 4.2 and 4.5)
- Combining symbolic execution of JavaScript with automatic GUI exploration and other needed components, we build the first end-to-end automated system for exploration of client-side JavaScript. (Section 4.2)

- We demonstrate the practical use of our implementation by applying it to automatically discovering 11 client-side script injection vulnerabilities, including two that were previously unknown. (Section 4.6)

4.1 Problem Statement and Overview

In this section we state the problem we focus on, exploring the execution space of JavaScript applications; describe one of its applications, finding client-side script injection vulnerabilities; and give an overview of our approach.

Problem statement. We develop techniques to systematically explore the execution space of JavaScript application code.

JavaScript applications often take many kinds of input. We view the input space of a JavaScript program as split into two categories: the *event space* and the *value space*.

- *Event space.* Rich web applications typically define tens to hundreds of JavaScript event handlers, which may execute in any order as a result of user actions such as clicking buttons or submitting forms. Event handler code may check the state of GUI elements (such as check-boxes or selection lists). The ordering of events and the state of the GUI elements together affects the behavior of the application code.

- *Value space.* The values of inputs supplied to a program also determine its behavior. JavaScript has numerous interfaces through which input is received:

  - *User data.* Form fields, text areas, and so on.
  - *URL and cross-window communication abstractions.* Web principals hosted in other windows or frames can communicate with JavaScript code via inter-frame communication abstractions such as URL fragment identifiers and HTML 5's proposed `postMessage`, or via URL parameters.
  - *HTTP channels.* Client-side JavaScript code can exchange data with its originating web server using `XMLHttpRequest`, HTTP cookies, or additional HTTP `GET` or `POST` requests.

This chapter primarily focuses on techniques to systematically explore the value space using symbolic execution of JavaScript, with the goal of generating inputs that exercise new program paths. However, automatically exploring the event space is also required to achieve good coverage. To demonstrate the efficacy of our techniques in an end-to-end system, we combine symbolic execution of JavaScript for the value space with a GUI exploration technique for the event space. This full system is able to automatically explore the combined input space of client-side web application code.

Application: finding client-side script injection vulnerabilities. Exploring a program's execution space has a number of applications in the security of client-side web appli-

cations. In this chapter, we focus specifically on one security application, finding client-side script injection vulnerabilities.

As with Flax, we treat all URLs and cross-window communication abstractions as untrusted sources, as such inputs may be controlled by an untrusted web principal. In addition, we also treat user data as an untrusted source because we aim to find cases where user data may be interpreted as code. The severity of attacks from user-data on client-side is often less severe than a remote XSS attack, but developers tend to fix these and Kudzu takes a conservative approach of reporting them. HTTP channels such as `XMLHttpRequest` are currently restricted to communicating with a web server from the same domain as the client application, so we do not treat them as untrusted sources. Developers may wish to treat HTTP channels as untrusted in the future when determining susceptibility to cross-channel scripting attacks [18], or when enhanced abstractions (such as the proposed cross-origin XMLHttpRequest [119]) allow cross-domain HTTP communication directly from JavaScript.

To effectively find XSS vulnerabilities, we require two capabilities: (a) generating directed test cases that explore the execution space of the program, and (b) checking, on a given execution path, whether the program validates all untrusted data sufficiently before using it in a critical sink. Custom validation checks and parsing routines are the norm rather than the exception in JavaScript applications, so our tool must check the behavior of validation rather than simply confirming that it is performed.

In previous work, we developed a tool called Flax which employs taint-guided fuzzing for finding client-side script injection attacks [100]. However, Flax relies on an external, manually developed test harness to explore the path space. Kudzu, in contrast, automatically generates a test suite that explores the execution space systematically. Kudzu also uses symbolic reasoning (with its constraint solver) to check if the validation logic employed by the application is sufficient to block malicious inputs — this is a one-step mechanism for directed exploit generation as opposed to multiple rounds of undirected fuzzing employed in Flax. Static analysis techniques have also been employed for JavaScript [47] to reason about multiple paths, but can suffer from false positives and do not produce test inputs or attack instances. Symbolic analyses and model-checking have been used for server-side code [75, 8]; however, the complexity of path conditions we observe requires more expressive symbolic reasoning than supported by tools for server-side code.

Approach Overview. The value space and event space of a web application are two different components of its input space: code reachable by exploring one part of the input space may not be reachable by exploring the other component alone. For instance, exploring the GUI event space results in discovering new views of the web application, but this does not directly affect the coverage that can be achieved by systematically exploring all the paths in the code implementing each view. Conversely, maximizing path coverage is unlikely to discover functionality of the application that only happens when the user explores a different application view. Therefore, Kudzu employs different techniques to explore each part of the input space independently.

Value space exploration. To systematically explore different execution paths, we develop

a component that performs dynamic symbolic execution of JavaScript code, and a new constraint solver that offers the desired expressiveness for automatic symbolic reasoning.

In dynamic symbolic execution, certain inputs are treated as symbolic variables. Dynamic symbolic execution differs from normal execution in that while many variable have their usual (*concrete*) values, like 5 for an integer variable, the values of other variables which depend on symbolic inputs are represented by *symbolic* formulas over the symbolic inputs, like $input_1 + 5$. Whenever any of the operands of a JavaScript operation is symbolic, the operation is simulated by creating a formula for the result of the operation in terms of the formulas for the operands. When a symbolic value propagates to the condition of a branch, KUDZU can use its constraint solver to search for an input to the program that would cause the branch to make the opposite choice.

Event space exploration. As a component of KUDZU we develop a GUI explorer that searches the space of all event sequences using a random exploration strategy. KUDZU's GUI explorer component randomly selects an ordering among the user events registered by the web page, and automatically fires these events using an instrumented version of the web browser. KUDZU also has an input-feedback component that can replay the sequence of GUI events explored in any given run, along with feeding new values generated by the constraint solver to the application's data inputs.

Testing for client-side script injection vulnerabilities. For each input explored, KUDZU determines whether there is a flow of data from an untrusted data source to a critical sink. If it finds one, it seeks to determine whether the program sanitizes and/or validates the input correctly to prevent attackers from injecting dangerous elements into the critical sink. Specifically, it attempts to prove that the validation is insufficient by constructing an attack input. As we will describe in more detail in Section 4.2, it combines the results of symbolic execution with a specification for attacks to create a constraint solver query. If the constraint solver finds a solution to the query, it represents an attack that can reach the critical sink and exploit a client-side script injection vulnerability.

4.2 End-to-End System Design

This section describes the various components that work together to make a complete KUDZU-based vulnerability-discovery system work. The full explanation of the constraint solver is in Sections 4.3 through 4.5. For reference, the relationships between the components are summarized in Figure 6.4.

System Components

First, we discuss the core components that would be used in any application of KUDZU: the *GUI explorer* that generates input events to explore the event space, the *dynamic symbolic interpreter* that performs symbolic execution of JavaScript, the *path constraint extractor* that builds queries based on the results of symbolic execution, the *constraint solver* that

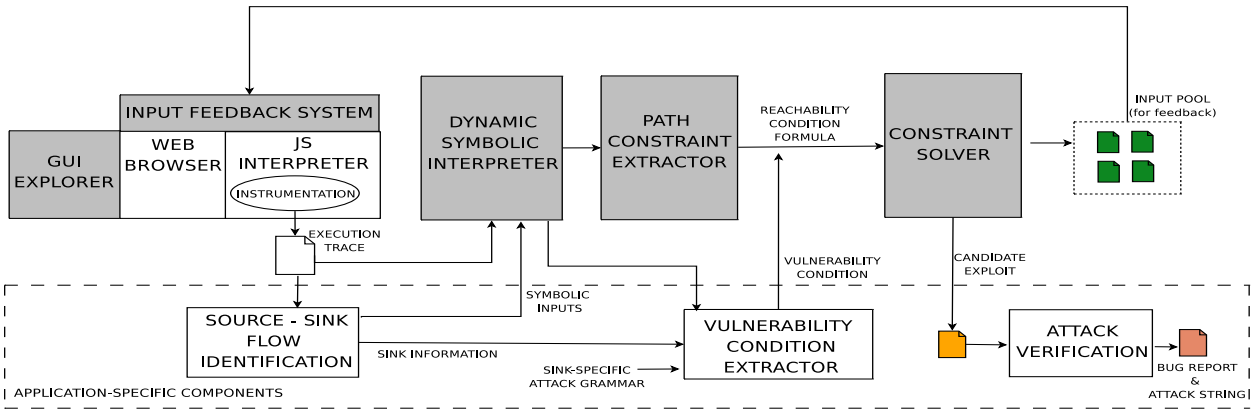

Figure 4.1: Architecture diagram for KUDZU. The components drawn in the dashed box perform functions specific to our application of finding client-side script injection. The remaining components are application-agnostic. Components shaded in light gray are the core contribution of this chapter.

finds satisfying assignments to those queries, and the *input feedback* component that uses the results from the constraint solver as new program inputs.

The GUI explorer. The first step in automating JavaScript application analysis is exploring the event space of user interactions. Each event corresponds to a user interaction such as clicking a check-box or a button, setting focus on a field, adding data to data fields, clicking a link, and so on. KUDZU currently explores the space of all sequences of events using a random exploration strategy. One of the challenges is to comprehensively detect all events that could result in JavaScript code execution. To address this, KUDZU instruments the browser functions that process HTML elements on the current web page to record when an event handler is created or destroyed. KUDZU's GUI explorer component randomly selects an ordering among the user events registered by the web page and executes them. [1] The random seed can be controlled to replay the same ordering of events. While invoking handlers, the GUI component also generates (benign) random test strings to fill text fields. (Later, symbolic execution will generate new input values for these fields to explore the input space further.) Links that navigate the page away from the application's domain are cancelled, thereby constraining the testing to a single application domain at a time. Investigating alternative strategies to prioritize the execution of events is a promising direction for further improvement.

Dynamic symbolic interpreter. KUDZU performs dynamic symbolic execution by first recording an execution of the program with concrete inputs, and then symbolically interpreting the recorded execution in a dynamic symbolic interpreter. For recording an execution

[1] Invoking an event handler may invalidate another handler (for instance, when the page navigates as a result). In that case, the invalidated handlers are ignored and if new handlers are created by the event that causes invalidation, these events are explored subsequently.

trace, KUDZU employs an existing instrumentation component [100] implemented in the web browser's JavaScript interpreter. For each JavaScript bytecode instruction executed, it records the semantics of the operation, its operands and operand values in a simplified intermediate language called JASIL[100]. The set of JavaScript operations captured includes all operations on integers, booleans, strings, arrays, as well as control-flow decisions, object types, and calls to browser-native methods. For the second step, dynamic symbolic execution, we have developed from scratch a symbolic interpreter for the recorded JASIL instructions.

Symbolic inputs for KUDZU are configurable to match the needs of an application. For instance, in the application we consider, detecting client-side script injection, all URL data, data received over cross-window communication abstractions, and user data fields are marked symbolic. Symbolic inputs may be strings, integers, or booleans. Symbolic execution proceeds on the JASIL instructions in the order they are recorded in the execution trace. At any point during dynamic symbolic execution, a given operand is either *symbolic* or *concrete*. If the operand is symbolic, it is associated with a *symbolic value*; otherwise, its value is purely concrete and is stored in the dynamic execution trace. When interpreting a JASIL operation in the dynamic symbolic interpreter, the operation is symbolically executed if one or more of its input operands is symbolic. Otherwise the operation of the symbolic interpreter on concrete values would be exactly the same as the real JavaScript interpreter, so we simply reuse the concrete results already stored in the execution trace.

The symbolic value of an operand is a formula that represents its computation from the symbolic inputs. For instance, for the JASIL assignment operation `y := x`, if x is symbolic (say, with the value $input_1 + 5$), then symbolic execution of the operation copies this value to `y`, giving `y` the same symbolic value. For an arithmetic operation, say `y := x1 + x2` where `x1` is symbolic (say with value $input_2 + 3$) and `x2` is not (say with the concrete value 7), the symbolic value for `y` is the formula representing the sum ($input_2$+10). Operations over strings and booleans are treated in the same way, generating formulas that involve string operations like `match` and boolean operations like `and`. At this point, string operations are treated simply as uninterpreted functions. During the symbolic execution, whenever the symbolic interpreter encounters an operation outside the supported formula grammar, it forces the destination operand to be concrete. For instance, if the operation is `x = parseFloat(s)` for a symbolic string `s`, `x` and `s` can be replaced with their concrete values from the trace (say, 4.3 and `'' 4.3''`). This allows symbolic computation to continue for other values in the execution.

Path constraint extractor. The execution trace records each control-flow branch (e.g., `if` statement) encountered during execution, along with the concrete value (true or false) representing whether the branch was taken. During symbolic execution, the corresponding *branch condition* is recorded by the path constraint extractor if it is symbolic. As execution continues, the formula formed by conjoining the symbolic branch conditions (negating the conditions of branches that were not taken) is called the *path constraint*. If an input value satisfies the path constraint, then the program execution on that input will follow the same

execution path.

To explore a different execution path, KUDZU selects a branch on the execution path and builds a modified path constraint that is the same up to that branch, but that has the negation of that branch condition (later conditions from the original branch are omitted). An input that satisfies this condition will execute along the same path up to the selected branch, and then explore the opposite alternative. There are several strategies for picking the order in which branch conditions can be negated—KUDZU currently uses a generational search strategy [41].

Constraint solver. Most symbolic execution tools in the past have relied on an existing constraint solver. However, KUDZU generates a rich set of constraints over string, integer and boolean variables for which existing off-the-shelf solvers are not powerful enough. Therefore, we have built a new solver, KALUZA, for our constraints (we present the algorithm and design details in Section 4.4). In designing this component, we examined the symbolic constraints KUDZU generates in practice. From the string constraints arising in these, we distilled a set of primitive operations required in a core constraint language. (This core language is detailed in Section 4.3, while the solver's full interface is given in Section 4.5.) We justify our intuition that solving the core constraints is sufficient to model JavaScript string operations in Section 4.5, where we show a practical translation of JavaScript string operations into our constraint language.

Input feedback. Solving the path constraint formula using the solver creates a new input that explores a new program path. These newly generated inputs must be fed back to the JavaScript program: for instance simulated user inputs must go in their text fields, and GUI events should be replayed in the same sequence as on the original run. The input feedback component is responsible for this task. As a particular HTML element (e.g a text field) in a document is likely allocated a different memory address on every execution, the input feedback component uses XPath [128] and DOM identifiers to uniquely identify HTML elements across executions and feed appropriate values into them. If an input comes from an attribute for a DOM object, the input feedback component sets that attribute when the object is created. If the input comes via a property of an event that is generated by the browser when handling cross-window communication, such as the `origin` and `data` properties of a `postMessage` event, the component updates that property when the JavaScript engine accesses it. KUDZU instruments the web browser to determine the context of accesses, to distinguish between accesses coming from the JavaScript engine and accesses coming from the browser core or instrumentation code.

Application-specific components

Next, we discuss three components that are specialized for the task of finding client-side script injection vulnerabilities: a *sink-source identification* component that determines which critical sinks might receive untrusted input, a *vulnerability condition extractor* that captures domain knowledge about client-side script injection attacks, and the *attack verification* com-

ponent that checks whether inputs generated by the tool in fact represent exploits.

Sink-source identification. To identify if external inputs are used in critical sink operations such as `eval` or `document.write`, we perform a dynamic data flow analysis on the execution trace. As outlined earlier, we treat all URL data, data received over cross-window communication abstractions (such as `postMessage`), and data filled into user data fields as potentially untrusted. The data flow analysis is similar to a dynamic taint analysis. Any execution trace that reveals a flow of data to a critical sink is subject to further symbolic analysis for exploit generation. We use an existing framework, FLAX, for this instrumentation and taint-tracking [100] in a manner that is faithful to the implementation of JavaScript in the WebKit interpreter.

Vulnerability condition extractor. An input represents an attack against a program if it passes the program's validation checks, but nonetheless implements the attacker's goals (i.e., causes a client-side script injection attack) when it reaches a critical sink. The vulnerability condition extractor collects from the symbolic interpreter a formula representing the (possibly transformed) value used at a critical sink, and combines it with the path constraint to create a formula describing the program's validation of the input.[2]. To determine whether this value constitutes an attack, the vulnerability condition extractor applies a sink-specific vulnerability condition specification, which is a (regular) grammar encoding a set of strings that would constitute an attack against a particular sink. This specification is conjoined with the formula representing the transformed input to create a formula representing values that are still dangerous after the transformation.

For instance, in the case of the `eval` sink, the vulnerability specification asserts that a valid attack should be zero or more statements each terminated by a ';', followed by the payload. Such grammars can be constructed by using publicly available attack patterns [94]. The tool's attack grammars are currently simple and can be extended easily for comprehensiveness and to incorporate new attacks.

To search only for realistic attacks, the specification also incorporates domain knowledge about the possible values of certain inputs. For instance, when a string variable corresponds to the web URL for the application, we assert that the string starts with the same domain as the application.

Attack verification. KUDZU automatically tests the exploit instance by feeding the input back to the application, and checking if the attack payload (such as a script with an alert message) is executed. If this verification fails, KUDZU does not report an alarm.

[2]Sanitization for critical client-side sink operations may happen on the server side (when data is sent back via `XMLHttpRequest`.) Our implementation handles this by recognizing such transformations using approximate tainting techniques [100] for data transmitted over `XMLHttpRequest`

4.3 Core Constraint Language

In order to support a rich language of input constraints with a simple solving back end, we have designed an intermediate form we call the *core constraint language*. This language is rich enough to express constraints from JavaScript programs, but simple enough to make solving the constraints efficient. In this section we define the constraint language, analyze its expressiveness and the theoretical complexity of deciding it, and compare its expressiveness to the core languages of previous solvers.

Language Definition

The abstract syntax for our core constraint language is shown in Figure 4.2. A formula in the language is an arbitrary boolean combination of constraints. Variables which represent strings may appear in five types of constraints. The first three constraint types indicate that a string is a member of the language defined by a regular expression, that two strings are equal, or one string is equal to the concatenation of two other strings. The two remaining constraints relate the length of one string to a constant natural number, or to the length of another string, by any of the usual equality or ordering operations. Regular expressions are formed from characters or the empty string (denoted by ϵ) via the usual operations of concatenation (represented by juxtaposition), alternation ($|$), and repetition zero or more times (Kleene star $*$).

The constraints all have their usual meanings. Any number of variables may be introduced, and *Character*s are drawn from an arbitrary non-empty alphabet, but *Number*s must be non-negative integers. For present purposes, strings may be of unbounded length, though we will introduce upper bounds on their lengths later.

Expressiveness and Complexity

Though the core constraint language is intentionally small, it is not minimal: some types of constraints are included for the convenience of translating to and from the core language, but do not fundamentally increase its expressiveness. String equality, comparisons between lengths and constants, and inequality comparisons between lengths can be expressed using concatenation, regular expressions, and equality between string lengths respectively; the details are omitted for space.

Each of the remaining constraint types (regular expression membership, concatenation, and length) makes its own contribution to the expressiveness of the core constraints. Our conference paper gives examples of the sets of strings that each constraint type can uniquely define [99]. The core constraint language is expressive enough that the complexity of deciding it is not known precisely; it is at least PSPACE-hard. These relationships are summarized in Figure 4.3. The complexity of our core constraint language falls to NP-complete when the lengths of string variables are bounded, as they are in our implementation. Further details are in the appendix.

$$
\begin{array}{rcl}
\mathit{Formula} & ::= & \neg \mathit{Formula} \\
& | & \mathit{Formula} \wedge \mathit{Formula} \\
& | & \mathit{Constraint} \\
\mathit{Constraint} & ::= & \mathit{Var} \in \mathit{RegExp} \\
& | & \mathit{Var} = \mathit{Var} \\
& | & \mathit{Var} = \mathit{Var} \circ \mathit{Var} \\
& | & \texttt{length}(\mathit{Var})\ \mathit{Rel}\ \mathit{Number} \\
& | & \texttt{length}(\mathit{Var})\ \mathit{Rel}\ \ \texttt{length}(\mathit{Var}) \\
\mathit{RegExp} & ::= & \mathit{Character} \\
& | & \epsilon \\
& | & \mathit{RegExp}\ \mathit{RegExp} \\
& | & \mathit{RegExp} \mid \mathit{RegExp} \\
& | & \mathit{RexExp}^* \\
\mathit{Rel} & ::= & < \mid \leq \mid = \mid \geq \mid >
\end{array}
$$

Figure 4.2: Abstract grammar of the core constraint language.

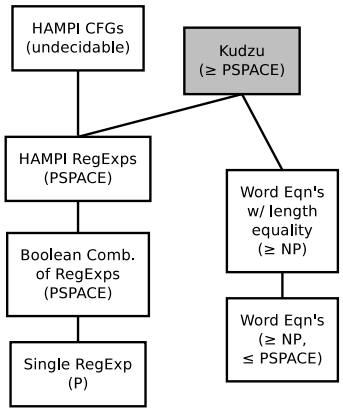

Figure 4.3: Relations between the unbounded versions of several theories of strings. Theories higher in the graph are strictly more expressive but are also at least as complex to decide. KUDZU's core constraint language (shaded) is strictly more expressive than either the core language of HAMPI [66] or the theory of word equations and an equal length predicate (the "pure library language" of [17]).

Expressiveness Comparison

Our system's core constraint language is more expressive than the constraints used in similar previous systems, and this expressiveness is key in allowing it to handle a more complex class of applications.

Bjørner *et al.* [17] present a "pure library language" that, like our core constraint lan-

guage, includes word equations and the ability to assert that two strings have the same length, so like our language its decidability is open. However, their language does not include regular expressions. Regular expressions may be less common in the .NET applications Bjørner *et al.* study, but they are used ubiquitously in JavaScript, so regular expression support is mandatory in our domain. Similarly the work of Caballero *et al.* [11, 22] deals with programs compiled from C-family languages, whose string operations are much more limited.

The DPRLE tool [50] focuses on a class of constraints that combine concatenation and regular expression matching, but in a more limited way than our tool supports. DPRLE addresses a different problem domain, since it gives solutions for constraints over languages (potentially infinite sets of strings) rather than single strings, but this makes the task significantly more difficult. We were unable to express the constraints from our application in DPRLE's input format or any straightforward extension of it. For instance, there is no way to express the constraint that two language expressions should be equal, not surprising since such constraints in general are undecidable [24].

HAMPI [66] provides support for regular expression constraints (and in fact we build on its implementation for this feature), but its support for other constraints is limited, particularly by the fact that it supports only a single string variable. The variable can be concatenated with constant strings, but these string expressions cannot be compared with each other, only with regular expressions, so HAMPI lacks the full generality of word equations. For instance, HAMPI constraints cannot define the set $\{uv\#u\#v : u, v \in \{0,1\}^*\}$.

It is worth reemphasizing that these limitations are not just theoretical: they make these previous systems unsuitable for our applications. One of the most common operations in the programs we examine is to parse a single input string (such as a URL) into separate input variables using `split` or repeated regular expression matching. Representing the semantics of such an operation requires relating the contents of one string variable to those of another, something that neither DPRLE nor HAMPI supports.

4.4 Core Constraint Solving Approach

We have implemented a decision procedure called Kaluza for the core set of constraints, which is available online [63]. In this section, we explain our algorithm for solving the core set of constraints. We introduce a bounded version of the constraints where we assume a user-supplied upper bound k on the length of the variables. This allows us to employ a SAT-based solution strategy without reducing the practical expressiveness of the language.

The algorithm satisfies three important properties, whose informal proof appears in the appendix.

1. *Soundness.* Whenever the algorithm terminates with an assignment of values to string variables, the solution is consistent with the input constraints.

2. *Bound-k completeness.* If there exists a solution for the string variables where all strings have length k or less, then our algorithm finds one such solution.

```
Input: C : constraint list
Output: (IsSat : bool, Solutions : string list)
G ← BuildConcatGraph(C);
(C′, StrOrderMap) ← DecideOrder(G);
C ← C ∪ C′;
FailLenDB : length_assignment list ← ∅;
while true do
    (X, lengths) ← SolveLengths(C, FailLenDB);
    if (X = UNSAT) then
        print `` Unsatisfiable'';
        halt(false, ∅);
    end
    Final : bitvector_constraints;
    Final ← CreateBVConstraints(StrOrderMap, C, lengths);
    (Result, BVSolutions) ← BVSolver(Final);
    if (Result = SAT) then
        print `` Satisfiable'';
        printSolutions(BVSolutions, lengths, StrOrderMap);
        halt(true, BVsToStrings(BVSolutions));
    end
    else
        FailLenDB ← FailLenDB ∪ lengths;
    end
end
```

Figure 4.4: Algorithm for solving the core constraints.

3. *Termination.* The algorithm requires only a finite number of steps (a function of the bound) for any input.

The solver translates constraints on the contents of strings into bit-vector constraints that can be solved with a SAT-based SMT solver. For this purpose, the solver translates each input string into a sequence of n-bit integers ($n = 8$ in the current implementation). Each string variable S also has an associated integer variable L_S representing its length. A single string is converted to a bit-vector by concatenating the binary representations of each character. Then, the bit-vectors representing each string are themselves concatenated into a single long bit-vector. (The order in which the strings are concatenated into the long vector reflects the concatenation constraints, as detailed in step 1 below.) The solver passes the constraints over this bit vector to a SMT (satisfiability modulo theories) decision procedure for the theory of bit vectors, STP [39] in our implementation. Informally, it is convenient to refer to the combined bit vector as if it were an array indexed by character offsets, but we do not use STP's theory of arrays, and character offsets are multiplied by n to give bit offsets before producing the final constraints.

Our algorithm is shown in Figure 4.4. At a high level, it has three steps. First, it translates string concatenation constraints into a layout of variables (with overlap) in the

final character array mentioned above. Second, it extracts integer constraints on the lengths of strings and finds a satisfying length assignment using the SMT solver. Finally, given a position and length for each string, the solver translates the constraints on the contents of each string into bit-vector constraints and checks if they are satisfiable.

In general, because of the interaction of length constraints and regular expressions, the length assignment chosen in step 2 might not correspond to satisfiable contents constraints, even when a different length assignment would. So if step 3 fails to find a satisfying assignment, the algorithm returns to step 2 to generate a new length assignment (distinct from any tried previously). Steps 2 and 3 repeat until the solver finds a satisfying assignment, or it has tried all possible length assignments (up to the length bound k).

Step 1: Translating concatenation constraints.. The intuition behind KUDZU's handling of concatenation constraints is that for a constraint $S_1 = S_2 \circ S_3$, it would be sufficient to ensure that S_2 comes immediately before S_3 in the final character array, and to lay out S_1 as overlapping with S_2 and S_3 (so that S_1 begins at the same character as S_2 and ends at the same character as S_3). This overlapping layout also has the advantage of reducing the total length of bit-vectors required. Each concatenation constraint suggests an ordering relation among the string variables, but it might not be possible to satisfy all such ordering constraints simultaneously.

To systematically choose an ordering, the solver builds a graph of concatenation constraints (a *concat graph* for short). The graph has a node for each string variable, and for each constraint $S_1 = S_2 \circ S_3$, S_2 and S_3 are the left and right children (respectively) of S_1. An example of such a graph is shown in Figure 4.5. Without loss of generality, we can assume that the graph is acyclic: if there is a cycle from S_1 to S_2 to S_3 ... back to S_1, then $S_1 = S_2 \circ S_3 \circ \cdots \circ S_1$ (or some other order), so all the variables other than S_1 must be the empty string, and can be removed from the constraints. (In our applications the constraints will in any case be acyclic by construction.) Given this graph, the algorithm then chooses the relative ordering of the strings in the character array by assigning start and end positions to each node with a post-order traversal of the graph. (In Figure 4.5, these positions are shown in parentheses next to each node.)

For the layout generated by the algorithm to be correct, the concat graph must be a DAG in which each internal node has exactly two children, and those children are adjacent in the layout. (This implies that the graph is planar.) The graph may not have these properties at construction; for instance, Figure 4.6 gives a set of example constraints with contradictory ordering requirements: S_2 cannot be simultaneously to the left and to the right of S_3. The algorithm resolves such requirements by duplicating a subtree of the graph (for instance as shown in the right half of Figure 4.6). To maintain the correct semantics, the algorithm adds string equality constraints to ensure that any duplicated strings have the same contents as the originals. The algorithm performs duplications to ensure that the graph satisfies the correctness invariant, but our current algorithm does not attempt to perform the minimal number of copies (for instance, in Figure 4.6 it would suffice to copy either only S_2 or only S_3), which in our experience has not hurt the solver's performance.

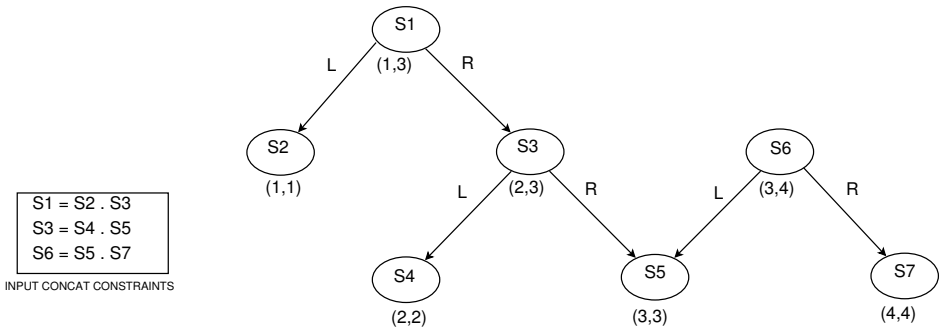

Figure 4.5: A sample concat graph for a set of concatenation constraints. The relative ordering of the strings in the final character array is shown as start and end positions in parentheses alongside each node.

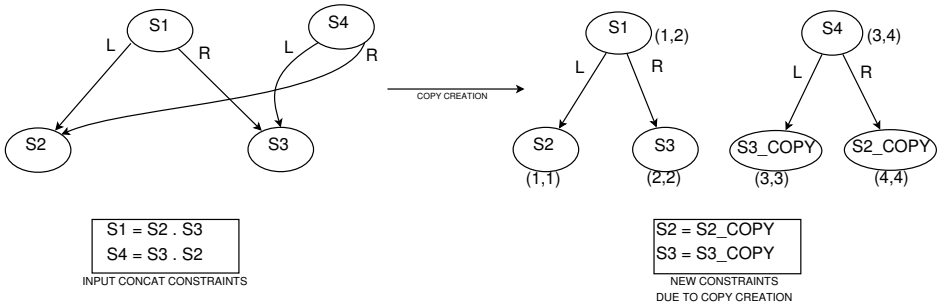

Figure 4.6: A set of concat constraints with contradictory ordering requirements. Nodes are duplicated to resolve the contradiction.

Step 2: Finding a satisfiable length assignment. Each string variable S has an associated length variable L_S. Each core string constraint implies a corresponding constraint on the lengths of the strings, as detailed in Table 4.1. For the regular expression containment constraint ($S_1 \in R$), the set of possible lengths is an *ultimately periodic set*: whether a length is possible depends only on its remainder when divided by a fixed period, except for a finite set of exceptions. (Yu *et al.* use the equivalent concept of a semi-linear set in a conservative automaton-based approach [135].) The details of computing this set are covered in the literature [77]; we note that such sets can be conveniently represented with our SMT solver since it supports a modulus operation. At each iteration of step 2, the solver conjoins the length constraints corresponding to all of the original string constraints, along with a constraint to rule out each length assignment that had previously been tried, and passes this formula to the SMT solver. If it returns a satisfying assignment, it represents a new length assignment to try; if the constraint is unsatisfiable, then so were the original string constraints.

It is not necessary for correctness that the length abstraction performed by the solver be precise, but determining precise length bounds improves performance by avoiding wasted iterations. In the complete system, the integer constraints over lengths are solved together

Core Constraint	Implication on lengths
$S_1 = S_2 \circ S_3$	$L_{S_1} = L_{S_2} + L_{S_3}$
$S_1 \in R$	$L_{S_1} \in \mathit{LengthSet}(R)$
$S_1 = S_2$	$L_{S_1} = L_{S_2}$
$\texttt{length}(S_1) \diamond i$	$L_{S_1} \diamond i$
$\texttt{length}(S_1) \diamond \texttt{length}(S_2)$	$L_{S_1} \diamond L_{S_2}$

Table 4.1: Length constraints implied by core string constraints, where L_S is the length of a string S, and $\diamond$ ranges over the operators $\{<, \leq, =, \geq, >\}$.

with integer constraints arising directly from the original program, discussed in Section 4.5. In our experience it is important for good performance to solve these two sets of integer constraints together. The two sets of constraints may be interrelated, and solving them together prevents the solver from wasting time exploring solutions that satisfy one set but not the other.

Step 3: Encoding as bitvectors. Given the array layout and lengths computed in steps 1 and 2, the remaining constraints over the contents of strings can be expressed as constraints over fixed-size bit-vectors. String equality translates directly into bit-vector equality. For the encoding of regular expression constraints, we reuse part of the implementation of the HAMPI tool [66]. At a high-level, the translation first unrolls uses of the Kleene star $*$ in a regular expression into a finite number of repetitions (never more than the string length). Next, where the regular expression has concatenation, HAMPI determines all possible combinations of lengths that sum to the total length, and instantiates each as a conjunction of constraints. Along with the alternations that appeared in the original regular expression, each of these conjunctions also represents an alternative way in which the regular expression could match the string. To complete the translation, the choice between all of these alternatives is represented with a disjunction. (See [66] for a more detailed explanation and some optimizations.)

HAMPI supports only a single, fixed-length input, so we invoke it repeatedly to translate each constraint between a regular expression and a string into an STP formula. We then combine each of these translations with our translations of other string contents constraints (e.g., string equality), and conjoin all of these constraints so that they apply to the same single long character array. It is this single combined formula that we pass to the SMT solver (STP) to find a satisfying assignment.

4.5 Reducing JavaScript to String Constraints

In this section we describe our tool's translation from JavaScript to the language of our constraint solver, focusing on the treatment of string operations. We start by giving the full constraint language the solver supports, then describe our general approach to modeling string operations, our use of concrete values from the dynamic trace, and the process of

$$
\begin{array}{rcl}
\tau & ::= & \mathit{string} \mid \mathit{int} \mid \mathit{bool} \\
\mathit{ConstRegex} & ::= & \mathit{Regex} \mid \mathit{CapturedBrack}(R, i) \\
 & & \mid \mathit{BetweenCapBrack}(R, i, j)
\end{array}
$$

Figure 4.7: Type system for the full constraint language

$$
\begin{array}{rcl}
S_1 : \mathit{string} & = & (S_2 : \mathit{string}) \circ (S_3 : \mathit{string}) \circ \cdots \\
I_1 : \mathit{int} & = & \texttt{length}(S : \mathit{string}) \\
S_1 : \mathit{string} & \in & R : \mathit{ConstRegex} \\
S_1 : \mathit{string} & \notin & R : \mathit{ConstRegex} \\
I_1 : \mathit{int} & = & (I_2 : \mathit{int})\ \{+, -, \cdot, /\}\ (I_3 : \mathit{int}) \\
B_1 : \mathit{bool} & = & (A_1 : \tau)\ \{=, \neq\}\ (A_2 : \tau) \\
B_1 : \mathit{bool} & = & (I_1 : \mathit{int})\ \{<, \leq, \geq, >\}\ (I_2 : \mathit{int}) \\
B_1 : \mathit{bool} & = & \neg(B_2 : \mathit{bool}) \\
B_1 : \mathit{bool} & = & (B_2 : \mathit{bool})\ \{\wedge, \vee\}\ (B_3 : \mathit{bool}) \\
S_1 : \mathit{string} & = & \texttt{toString}(I_1 : \mathit{int})
\end{array}
$$

Figure 4.8: Grammar and types for the full constraint language including operations on strings, integers, and booleans.

translating real regular expressions into textbook-style ones.

Full constraint language. The core constraint language presented in Section 4.3 captures the essence of our solving approach, but it excludes several features for simplicity, most notably integer constraints. The full constraint language supported by our solver supports values of string, integer, and boolean types, and its grammar is given in Figure 4.8, along with its type system in Figure 4.7. The additional constraints are solved at step 2 of the string solution procedure, together with the integer constraints on the lengths of strings. To match common JavaScript implementations (which reserve a bit as a type tag), we model integers as 31-bit signed bit-vectors in our SMT solver, which supports all the integer operations that JavaScript does. The solver replaces each `toString` constraint with the appropriate string once a value for its argument is selected: for instance, if i is given the value 12, $\texttt{toString}(i)$ is replaced with `‘‘ 12’’`.

JavaScript string operations. JavaScript has a large library of string operations, and we do not aim to support every operation, or the full generality of their behavior. Beyond the engineering challenge of building such a complete translation, having very complex symbolic translations for common operators would likely cause the system to bog down, and the generality would usually be wasted. Instead, our choice has been to model the string operations that occur commonly in web applications, and the core aspects of their behavior. For other operations and behavior aspects our tool uses values from the original execution trace (described further below), so that they are accurate with respect to the original execution even if the tool cannot reason symbolically about how they might change on modified executions. The detailed translation from several common operators (a subset of those supported by our implementation) to our constraint language is shown in Table 4.2.

Using dynamic information. One of the benefits of dynamic symbolic execution is that it provides the flexibility to choose between symbolic values (which introduce generality) and concrete values (which are less general, but guaranteed to be precise) to control the scope of the search process. Our tool's handling of string constraints takes advantage of concrete

values from the dynamic traces in several ways. An example is string `replace`, which is often used in sanitization to transform unsafe characters into safe ones. Our translation uses a concrete value for the number of occurrences of the searched-for pattern: if a pattern was replaced six times in the original run, the tool will search for other inputs in which the pattern occurs six times. This sacrifices some generality (for instance, if a certain attack is only possible when the string appears seven times). However, we believe this is a beneficial trade-off since it allows our tool to analyze and find bugs in many uses of `replace`. For comparison, most previous string constraint solvers do not support `replace` at all, and adding a `replace` that applied to any number of occurrences of a string (even limited to single-character strings) would make our core constraint language undecidable in the unbounded case [21].

Regular expressions in practice. The "regular expressions" supported by languages like JavaScript have many more features than the typical definition given in a computability textbook (or Figure 4.2). Figure 3.8 shows an example (adapted from a real web site) of one of many regular expressions Kudzu must deal with. Kudzu handles a majority of the syntax for regular expressions in JavaScript, which includes support for (possibly negated) character classes, escaped sequences, repetition operators (`{n}/?/*/+/`) and sub-match extraction using capturing parentheses. Kudzu keeps track of the nesting of capturing parentheses, so that it can express the relation between the input string and the parts of it that match the captured groups (as shown in Table 4.2). Kudzu does not currently support back-references (they are strictly more expressive than true regular expressions), though if we see a need in the future, many uses of back-references could be translated using (non-regular) concatenation constraints.

4.6 Experimental Evaluation

We have built a full-implementation of Kudzu using the WebKit browser, with 650, 7430 and 2200 lines of code in the path constraint extraction component, constraint solver, and GUI explorer component, respectively. The system is written in a mixture of C++, Ruby, and OCaml languages.

We evaluate Kudzu with three objectives. One objective is to determine whether Kudzu is practically effective in exploring the execution space of real-world applications and uncovering new code. The second objective is to determine the effectiveness of Kudzu as a stand-alone vulnerability discovery tool — whether Kudzu can automatically find client-side script injection vulnerabilities and prune away false reports. Finally, we measure the efficiency of the constraint solver. Our evaluation results are promising, showing that Kudzu is a powerful system that finds previously unknown vulnerabilities in real-world applications fully automatically.

JavaScript operation	Reduction to our constraint language
$S_1 : string = \text{charAt}(S : string, I : int)$	$(((L_S = \texttt{length}(S)) \wedge (I \geq 0) \wedge (I < L_S) \wedge (S = T_1 \circ T_2 \circ T_3) \wedge (T_2 = S_1) \wedge (I = \texttt{length}(T_1) + 1)) \vee (((I \geq L_S) \vee (I < 0)) \wedge (S_1 = \text{`` ''})))$
$I_1 : int = \text{charcodeAt}(S : string, I : int)$	$(((L_S = \texttt{length}(S)) \wedge (I \geq 0) \wedge (I < L_S) \wedge (S = T_1 \circ T_2 \circ T_3) \wedge (T_2 = S_1) \wedge (I = \texttt{length}(T_1) + 1) \wedge (S_1 = \texttt{toString}(I_1))) \vee (((I \geq L_S) \vee (I < 0)) \wedge (S_1 = \texttt{toString}(I_1)) \wedge (S_1 = \text{`` NaN''})))$
$S : string = \text{concat}(S_1 : string, S_2 : string, \ldots, S_k : string)$	$(S = S_1 \circ S_2 \circ \cdots \circ S_k)$
$I : int = \text{indexOf}(S : string, s : string, startpos : int)$	$(((I \geq 0) \wedge ((startpos < \texttt{length}(S)) \wedge (startpos \geq 0)) \wedge (S = S_1 \circ S_2) \wedge (startpos = \texttt{length}(S_1)) \wedge (S_2 = T_1 \circ s \circ T_3) \wedge (I = \texttt{length}(T_1)) \wedge (T_1 \notin Regex(\texttt{.*}s\texttt{.*}))) \vee ((I < 0) \wedge ((startpos < \texttt{length}(S)) \wedge (startpos \geq 0)) \wedge (S \notin Regex(\texttt{.*}s\texttt{.*}))) \vee (\neg((startpos < \texttt{length}(S)) \wedge (startpos \geq 0)) \wedge (S_1 = \texttt{toString}(I_1)) \wedge (S_1 = \text{`` NaN''})))$
$I : int = \text{lastIndexOf}(S : string, s : string, startpos : int)$	$(((I \geq 0) \wedge ((startpos < \texttt{length}(S)) \wedge (startpos \geq 0)) \wedge (S = S_1 \circ S_2) \wedge (startpos = \texttt{length}(S_1)) \wedge (S_2 = T_1 \circ s \circ T_3) \wedge (I = \texttt{length}(T_1)) \wedge (T_3 \notin Regex(\texttt{.*}s\texttt{.*}))) \vee ((I < 0) \wedge ((startpos < \texttt{length}(S)) \wedge (startpos \geq 0)) \wedge (S \notin Regex(\texttt{.*}s\texttt{.*}))) \vee (\neg((startpos < \texttt{length}(S)) \wedge (startpos \geq 0)) \wedge (S_1 = \texttt{toString}(I_1)) \wedge (S_1 = \text{`` NaN''})))$
$[S_1, S_2, \ldots, S_k] : string\ list = \text{match}(S : string, r : ConstRegex)$ (non-greedy)	$(((k > 0) \wedge (S \in r) \wedge ((S_1 \in CapturedBrack(r, 1)) \vee (S_1 = \text{`` ''})) \wedge (S = T_0 \circ S_1 \circ T_1 \circ \cdots \circ S_k \circ T_k) \wedge (\bigwedge_{i=0}^{k} T_i \in BetweenCapBrack(r, i, i+1))) \wedge \cdots \wedge ((S_k \in CapturedBrack(r, k)) \vee (S_k = \text{`` ''}))) \vee ((k \leq 0) \wedge (S \notin Regex(\texttt{.*}r\texttt{.*}))))$
$[S_1, S_2, \ldots, S_n] : string\ list = \text{match}(S : string, r : ConstRegex, n : int)$ (greedy match) n is the concrete number of occurrences of strings matching r.	$(((S = T_1 \circ M_1 \circ T_2 \circ M_2 \circ \cdots \circ T_n \circ M_n \circ T_{n+1}) \wedge (T_1, T_2, \ldots, T_{n+1} \notin Regex(\texttt{.*}r\texttt{.*})) \wedge (M_1, M_2, \ldots, M_n \in Regex(r)))$
$S_1 : string = \text{replace}(S : string, r : ConstRegex, s : string, n : int)$ n is the concrete number of occurrences of strings matching r in S.	$((S = T_1 \circ M_1 \circ T_2 \circ M_2 \circ \cdots \circ T_n \circ M_n \circ T_{n+1}) \wedge (T_1, T_2, \ldots, T_{n+1} \notin Regex(\texttt{.*}r\texttt{.*})) \wedge (M_1, M_2, \ldots, M_n \in Regex(r)))) \wedge (S_1 = T_1 \circ s \circ T_2 \circ s \circ \cdots \circ T_{n+1})$
$[S_1, S_2, \ldots, S_k] : string\ list = \text{split}(S : string, s : string, n : int)$ n is the concrete number of occurrences of strings matching r.	$(((S = S_1 \circ s \circ S_2 \circ s \circ \cdots \circ S_n \circ s \circ S_{n+1}) \wedge (S_1, S_2, \ldots, S_{n+1} \notin Regex(\texttt{.*}s\texttt{.*}))))$
$I_1 : int = \text{search}(S : string, r : ConstRegex)$	$(((I_1 < 0) \wedge (S \notin \texttt{.*}r\texttt{.*})) \vee ((I_1 \geq 0) \wedge (S = T_1 \circ T_2 \circ T_3) \wedge (I_1 = \texttt{length}(T_1)) \wedge (T_2 \in Regex(r)) \wedge (T_1, T_3 \notin (\texttt{.*}r\texttt{.*}))))$
$S_1 : string = \text{substring}(S : string, start : int, end : int)$	$((start \geq 0) \wedge (end < \texttt{length}(S)) \wedge (end \geq start) \wedge (S = T_1 \circ S_1 \circ T_2) \wedge (start = \texttt{length}(T_1)) \wedge (I_1 = end - start) \wedge (I_1 = \texttt{length}(S_1)) \vee ((start \geq 0) \wedge (end \geq \texttt{length}(S)) \wedge (end \geq start) \wedge (S = T_1 \circ S_1 \circ T_2) \wedge (start = \texttt{length}(T_1)) \wedge (L_S = \texttt{length}(S)) \wedge (I_1 = L_S - start) \wedge (I_1 = \texttt{length}(S_1)) \vee ((start < 0) \wedge (end < \texttt{length}(S)) \wedge (end \geq start) \wedge (S = T_1 \circ S_1 \circ T_2) \wedge (0 = \texttt{length}(T_1)) \wedge (I_1 = end - start) \wedge (I_1 = \texttt{length}(S_1)) \vee ((start < 0) \wedge (end \geq \texttt{length}(S)) \wedge (end \geq start) \wedge (S = T_1 \circ S_1 \circ T_2) \wedge (0 = \texttt{length}(T_1)) \wedge (L_S = \texttt{length}(S))(L_S = \texttt{length}(S_1))$
$B_1 : bool = \text{match}(S, r)$	$((B_1) \wedge (S \in Regex(r)) \vee (\neg(B_1) \wedge (S \notin Regex(r))))$
$I : int = \text{parseInt}(S)$	$(S = \texttt{toString}(I)) \wedge ((S = \text{`` NaN''}) \vee (S = Regex(\texttt{[0-9]+})))$

Table 4.2: Our reduction from common JavaScript operations to our full constraint language. Capitalized variables may be concrete or symbolic, while lowercase variables take a concrete value.

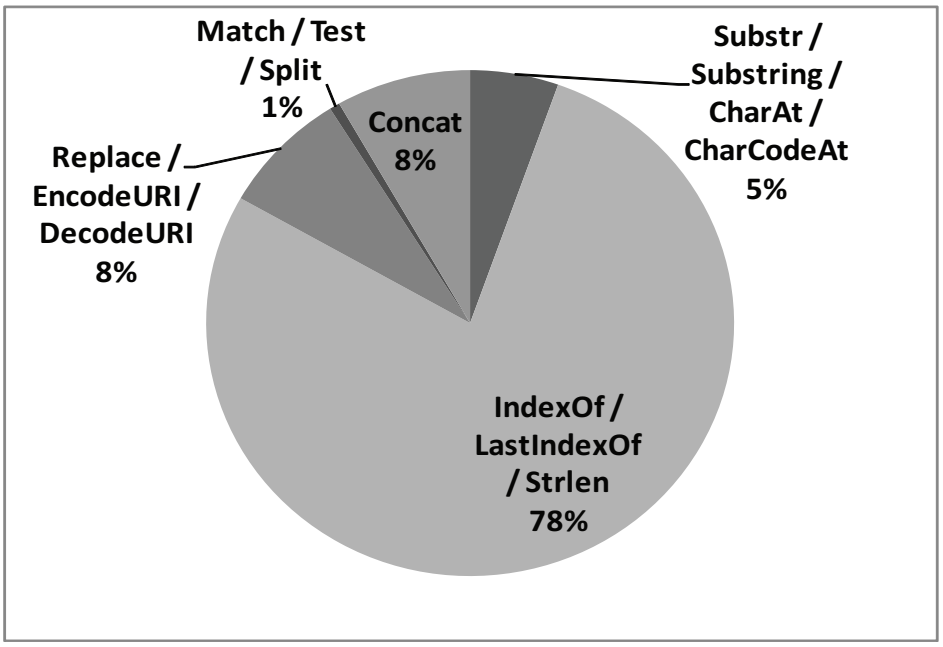

Figure 4.9: Distribution of string operations in our subject applications.

Experiment Setup

We select 18 subject applications consisting of popular iGoogle gadgets and AJAX applications, as these were studied by our previous tool Flax [100]. Flax assumes availability of an external (manually developed) test suite to seed its testing; in contrast, Kudzu automatically generates a much more comprehensive test suite and finds the points of vulnerability without requiring any external test harness a priori. Further, in our experiments Kudzu discovers 2 new vulnerabilities within a few hours of testing which were missed by the Flax because of its lack of coverage. In addition, as we show later in this section, many of the generated constraints are highly complex and not suitable for manual inspection or fuzzing, whereas Kudzu either asserts the safety of the validation checks or finds exploits for vulnerabilities in one iteration as opposed to many rounds of random testing.

To test each subject application, we seed the system with the URL of the application. For the gadgets, the URLs are the same as those used by iGoogle page to embed the gadget. We configure Kudzu to give a pre-prepared username and login password for applications that required authentication. We report the results for running each application under Kudzu, capping the testing time to a maximum of 6 hours for each application. All tests ran on a Ubuntu 9.10 Linux workstation with 2.2 GHz Intel dual-core processors and 2 GB of RAM.

Application	# of new inputs	Initial / Final Code Coverage	Bug found
Academia	20	30.27 / 76.47%	✓
AJAXIm	15	49.58 / 77.67%	✓
FaceBook Chat	54	26.85 / 76.84%	-
ParseUri	13	53.90 / 86.10%	✓
Plaxo	31	5.72 / 76.43%	✓
AskAWord	10	29.30 / 67.95 %	✓
Birthday Reminder	27	59.47 / 73.94%	-
Block Notes	457	65.06 / 71.50 %	✓
Calorie Watcher	16	64.54 / 73.53%	-
Expenses Manager	133	61.09 / 76.56%	-
Listy	19	65.31 / 79.73%	✓
NotesLP	25	46.62 / 76.67%	-
Progress Bar	12	63.60 / 75.09%	-
Simple Calculator	1	46.96 / 80.52%	✓
Todo List	15	72.51 / 86.41%	✓
TVGuide	6	30.39 / 75.13%	✓
Word Monkey	20	14.84 / 75.36%	✓
Zip Code Gas	11	59.05 / 74.28%	-
Average	**49**	**46.95 / 76.68%**	**11**

Table 4.3: The top 5 applications are AJAX applications, while the rest are Google/IG gadget applications. Column 2 reports the number of distinct new inputs generated, and column 3 reports the increase in code coverage from the initial run to and the final run.

Results

Table 4.3 presents the final results of testing the subject applications. The summary of our evaluation highlights three features of KUDZU: (a) it automatically discovers new program paths in real applications, significantly enhancing code coverage; (b) it finds 2 client-side script injection in the wild and several in applications that were known to contain vulnerabilities; and (c) KUDZU significantly prunes away false positives, successfully discarding cases that do employ sufficient validation checks.

Characteristics of string operations in our applications. Constraints arising from our applications have an average of 63 JavaScript string operations, while the remaining are boolean, logical and arithmetic constraints. Figure 4.9 groups the observed string operations by similarity. The largest fraction are operations like `indexOf` that take string inputs and return an integer, which motivate the need for a solver that reasons about integers and strings simultaneously. A significant fraction of the operations, including `subtring`, `split` and `replace`, implicitly give rise to new strings from the original one, thereby giving rise to constraints involving multiple string variables. Of the `match`, `split` and `replace` operations, 31% are regular expression based. Over 33% of the regular expressions have one or more capturing parentheses. Capturing parentheses in regular expression based match operations

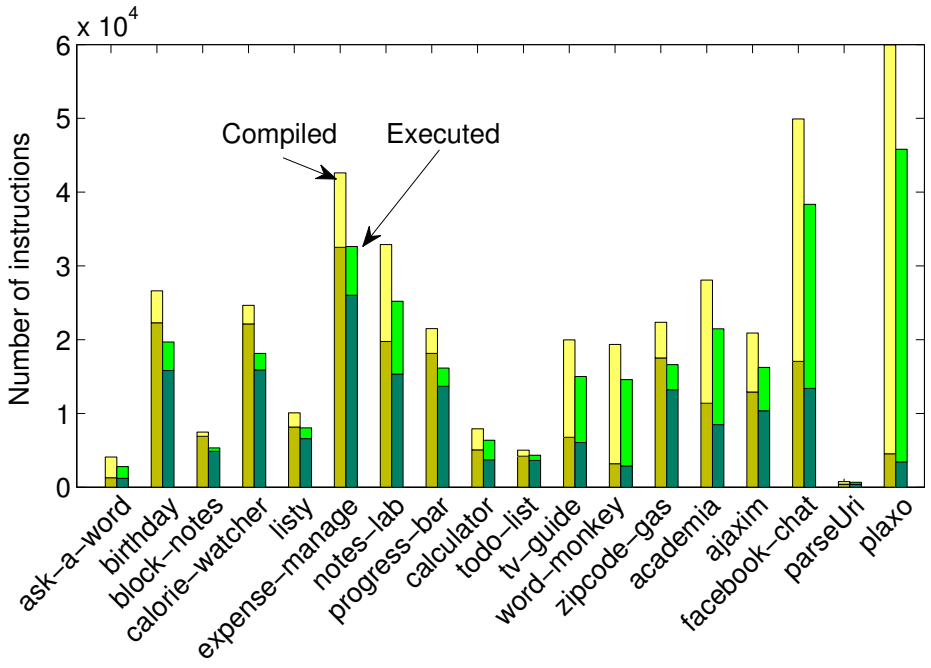

Figure 4.10: Kudzu code coverage improvements over the testing period. For each experiment, the right bar shows the increase in the executed code from the initial run to total code executed. The left bar shows the increase in the code compiled from initial run to the total code compiled in the entire test period.

lead to constraints involving multiple string variables, similar to operations such as `split`.

These characteristics show that a significant fraction of the string constraints arising in our target applications require a solver that can reason about multiple string variables. We empirically see examples of complex regular expressions as well as concatenation operations, which stresses the need for our solver that handles both word equations and regular expression constraints. Prior to this work, off-the-shelf solvers did not support word equations and regular expressions simultaneously.

Vulnerability Discovery. Kudzu is able to find client-side script injection vulnerabilities in 11 of the applications tested. 2 of these were not known prior to these experiments and were missed by Flax. One of them is on a social-networking application (`http://plaxo.com`) that was missed by our Flax tool because the vulnerability exists on a page linked several clicks away from the initial post-authentication page. The vulnerable code is executed only as part of a feature in which a user sets focus on a text box and uses it to update his or her profile. This is one of the many different ways to update the profile that the application provides. Kudzu found that only one of these ways resulted in a client-side script injection vulnerability, while the rest were safe. In this particular functionality, the application fails to properly validate a string from a `postMessage` event before using it in an `eval` operation. The application implicitly expects to receive this message from a window hosted at a sub-

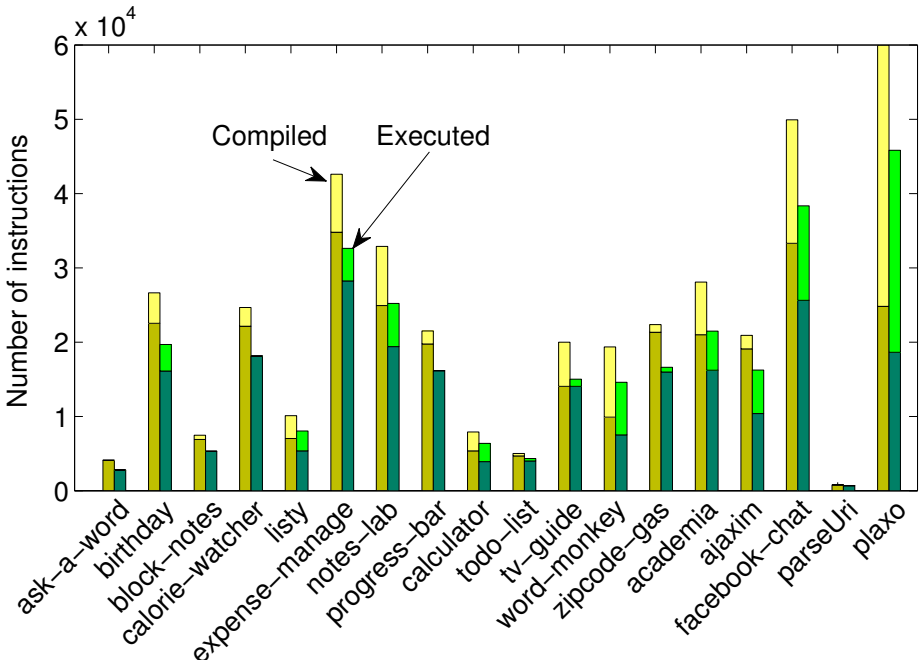

Figure 4.11: Benefits from symbolic execution alone (dark bars) vs. complete Kudzu (light bars). For each experiment, the right bar shows the increase in the total executed code when the event-space exploration is also turned on. The left bar shows the observed increase in the code compiled when the event-space exploration is turned on.

domain of `facebook.com`; however, KUDZU automatically determines that *any* web principal could inject any data string matching the format `FB_msg:.*{.*}`. This subsequently results in code injection because the vulnerable application fails to validate the origin of the sender and the structure of JSON string before its use in `eval`.

The second new vulnerability was found in a Google/IG gadget called Todo List. Similar to the previous case, the vulnerability becomes reachable only when a specific value is selected from a dropdown box. This interaction is among many that the gadget provides and we believe that KUDZU's automatic exploration is the key to discovering this use case. In several other cases, such as AjaxIM, the vulnerable code is executed only after several events are executed after initial sign-in page—Kudzu automatically reaches them during its exploration.

KUDZU did not find vulnerabilities in only one case that FLAX reported a bug. This is because the vulnerability was patched in the time period between our experimental evaluation of FLAX and Kudzu.

Code and Event-space Coverage. Table 4.3 shows the code coverage by executing the initial URL, and the final coverage after the test period. Measuring code coverage in a dynamically compiled language is challenging because all the application code is not known prior to the experiments. In our experiments, we measured the total code compiled during

our experiments and the total code executed.[3]

Table 4.3 shows an average improvement of over 29% in code coverage. The coverage varies significantly depending on the application. Figure 4.10 provides more detail. On several large applications, such as Facebook Chat, AjaxIM, and Plaxo, Kudzu discovers a lot of new code during testing. Kudzu is able to concretely execute several code paths, as shown by the increase in the right-side bars in Figure 4.10. On the other less complex gadget applications, most of the relevant code is observed during compilation in the initial run itself, leaving a relatively smaller amount of new code for Kudzu to discover. We also manually analyzed the source code of these applications and found that a large fraction of their code branches were not dependent on data we treat as untrusted.

To measure the benefits of symbolic execution alone, we repeated the experiments with the event-space exploration turned off during the test period and report the comparison to full-featured Kudzu in Figure 4.11. We consistently observe that symbolic execution alone discovers and executes a significant fraction of the application by itself. The event-exploration combined with symbolic execution does perform strictly better than symbolic execution in all but 3 cases. In a majority of the cases, turning on the event-space exploration significantly complements symbolic execution, especially for the AJAX applications which have a significant GUI component. In the 3 cases where improvements are not significant, we found that the event exploration generally either led to off-site navigations or the code executed could be explored by symbolic execution alone. For example, in the parseUri case, same code is executed by text-box input as well as by clicking a button on the GUI.

Table 4.4 shows the increase in number of events executed by Kudzu from the initial run to the total at the end of test period. These events include all keyboard and mouse events which result in execution of event handlers, navigation, form submissions and so on. We find that new events are dynamically generated during one particular execution as well as when new code is discovered. As a result, Kudzu gradually discovers new events and was able to execute approximately 50% of the events it observes during the period of testing.

Effectiveness. KUDZU automatically generates a test suite of 49 new distinct inputs on average for an application in the test period (shown in column 2 of table 4.3).

In the exploitable cases we observed, KUDZU was able to show the existence of a vulnerability with an attack string once it reached the point of vulnerability. That is, its constraint solver correctly determines the sufficiency or insufficiency of validation checks in a single query without manual intervention or undirected iteration. This eliminates false positives significantly in practice. For instance, KUDZU found that the Facebook web application has several uses of `postMessage` data in `eval` constructs, but all uses were correctly preceded by checks that assert that the origin of the message is a domain ending in `.facebook.com`. In contrast, the vulnerability in Plaxo fails to check this and KUDZU identifies the vulnerability

[3]One unit of code in our experiments is a JavaScript bytecode compiled by the interpreter. To avoid counting the same bytecode across several runs, we adopted a conservative counting scheme. We assigned a unique identifier to each bytecode based on the source file name, source line number, line offset and a hash of the code block (typically one function body) compiled.

Application	# of initial events fired	# of total events fired	Total events discovered
Academia	20	78	310
AJAXIm	72	481	988
FaceBook Chat	15	989	1354
ParseUri	5	16	17
Plaxo	88	381	688
AskAWord	2	8	11
Birthday Reminder	12	20	20
Block Notes	7	85	319
Calorie Watcher	14	18	22
Expenses Manager	10	107	1473
Listy	15	470	638
NotesLP	10	592	1034
Progress Bar	8	24	36
Simple Calculator	17	34	67
Todo List	8	26	61
TVGuide	17	946	1517
Word Monkey	3	10	22
Zip Code Gas	12	12	12
Average	**18.61**	**238.72**	**477.17**

Table 4.4: Event space Coverage: Column 2 and 3 show the number of events fired in the first run and in total. The last column shows the total events discovered during the testing.

the first time it reaches that point. Some of the validation checks Kudzu deals with are quite complex — Figure 3.8 shows an example which is simplified from a real application. These examples are illustrative of the need for automated reasoning tools, because checking the sufficiency of such validation checks would be onerous by hand and impractical by random fuzzing. Lastly, we point out that like most other vulnerability discovery tools, Kudzu can have false negatives because it may fail to cover code, or because of overly strict attack grammars.

Constraint Solver Evaluation. Figure 4.12 shows the running times for solving queries of various input constraint sizes. Each constraint is either a JavaScript string, arithmetic, logical, or boolean operation. The sizes of the equations varied from 1 to up to 250 constraints. The solver decides satisfiability of the constraints typically under a second for satisfiable cases. As expected, to assert unsatisfiability, the solver often takes time varying from nearly a second to 50 seconds. The variation is large because in many cases the solver asserts unsatisfiable by asserting the unsatisfiability of length constraints, which is inexpensive because the step of bit-vector encoding is avoided. In other cases, the unsatisfiability results only when the solver determines the unsatisfiability of bit-vector solutions.

Our solver requires only an upper bound on the lengths of input variables, and is able to infer satisfiable lengths of variables internally. In these experiments, we increase the upper bound of the input variables from 10 to 100 characters in steps of 20 each. If the solver asserts

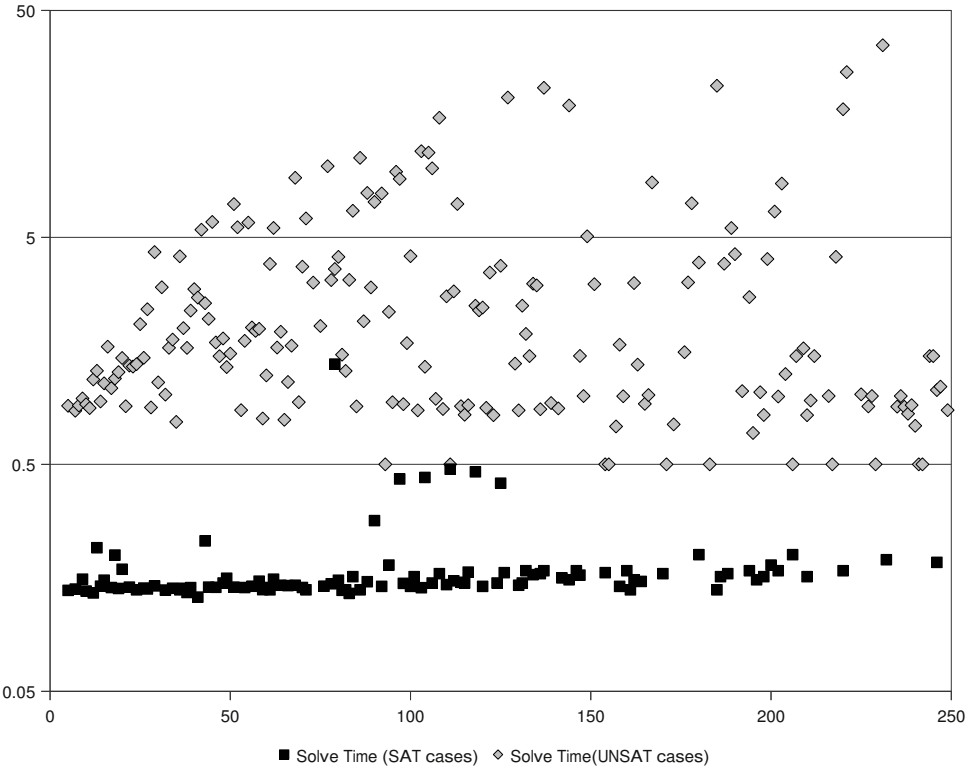

Figure 4.12: The constraint solver's running time (in seconds) as a function of the size of the input constraints (in terms of the number of symbolic JavaScript operations)

unsatisfiability up to the length bound of 100, the constraints are deemed unsatisfiable.

4.7 Related Work

Kudzu is the first application of dynamic symbolic execution to client-side JavaScript. Here, we discuss some related projects that have applied similar techniques to server-side web applications, or have used different techniques to search for JavaScript bugs. Finally, we summarize why we needed to build a new string constraint solver.

Server-side analysis. JavaScript application code is similar in some ways to server-side code (especially PHP); for instance, both tend to make heavy use of string operations. Several previous tools have demonstrated the use of symbolic execution for finding SQL injection and reflected or stored cross-site scripting attacks to code written in PHP and Java [122, 65, 5]. However, JavaScript code usually parses its own input, so JavaScript symbolic execution requires more expressive constraints, specifically to relate different strings that were previously part of a single string. Additional challenges unique to JavaScript arise

because JavaScript programs take many different kinds of input, some of which come via user interface events.

Like KUDZU, the Saner [8] tool for PHP aims to check whether sanitization routines are sufficient, not just that they are present. However their techniques are quite different: they select paths and model transformations statically, then perform testing to verify some vulnerabilities. Their definition of sanitization covers only string transformations, not validation checks involving branches, which occur frequently in our applications.

Analysis frameworks for JavaScript. Several works have recently applied static analysis to detect bugs in JavaScript applications (e.g., [47, 26]). Static analysis is complementary to symbolic execution: if a static analysis is sound, an absence of bug reports implies the absence of bugs, but static analysis warnings may not be enough to let a developer reproduce a failure, and in fact may be false positives.

FLAX uses taint-enhanced blackbox fuzzing to detect if the JavaScript application employs sufficient validation or not [100]; like KUDZU, it searches for inputs to trigger a failure. However, FLAX requires an external test suite to be able to reach the vulnerable code, whereas KUDZU generates a high-coverage test suite automatically. Also, FLAX performs only black-box fuzz testing to find vulnerabilities, while KUDZU's use of a constraint solver allows it to reason about possible vulnerabilities based on the analyzed code.

Crawljax is a recently developed tool for event-space exploration of AJAX applications [78]. Specifically, Crawljax builds a static representation of a Web 2.0 application by clicking elements on the page and building a state graph from the resulting transitions. KUDZU's value space exploration complements such GUI exploration techniques and enables a more complete analysis of the application using combined symbolic execution and GUI exploration.

String constraint solvers. String constraint solvers have recently seen significant development, and practical tools are beginning to become available, but as detailed in Section 4.3, no previous solvers would be sufficient for JavaScript, since they lack support for regular expressions [17, 11, 22], string equality [50], or multiple variables [66], which are needed in combination to reason about JavaScript input parsing. In concurrent work, Veanes *et al.* give an approach based on automata and quantified axioms to reduce regular expressions to the Z3 decision procedure [116]. Combined with [17], this would provide similar expressiveness to KUDZU.

4.8 Conclusion

With the rapid growth of AJAX applications, JavaScript code is becoming increasingly complex. In this regard, security vulnerabilities and analysis of JavaScript code is an important area of research. In this chapter, we presented the design of the first complete symbolic-execution based system for exploring the execution space of JavaScript programs. In making the system practical we addressed challenges ranging from designing a more expressive lan-

guage for string constraints to implementing exploration and replay of GUI events. We have implemented our ideas in a tool called Kudzu. Given a URL for a web application, Kudzu automatically generates a high-coverage test suite. We have applied Kudzu to find client-side script injection vulnerabilities and Kudzu finds 11 vulnerabilities (2 previously unknown) in live applications without producing false positives.

Chapter 5

Analysis of Existing Defenses

In this thesis so far, we have discussed techniques for finding scripting vulnerabilities in web applications. In this chapter, we study existing techniques that developers use to prevent these vulnerabilities from manifesting in the first place.

A central reason for the wide-spread prevalence of scripting attacks is the ad-hoc nature of output generation from web applications today. Web applications emit code intermixed with data in an unstructured way. *Web application output* is essentially text strings which can be emitted from the server-side code (in Java or PHP) or from client-side code in JavaScript. When a portion of the application output controlled by the attacker is parsed by the browser as a script, a script injection attack results.

Sanitization Defenses & Known Problems. The predominant first-line of defense against scripting vulnerabilities is *sanitization*—the process of applying encoding or filtering primitives, called *sanitization primitives* or *sanitizers*, to render dangerous constructs in untrusted inputs inert [8, 129, 110, 125]. There are two well-established problems known about the practice of manually applying sanitizers which make it is notoriously prone to manual errors [8, 97, 99, 100, 75, 62]. First, developers often implement sanitization primitives incorrectly [51, 8]. Second, developers often fail to apply *any* sanitization to untrusted content before embedded it inline in code that is parsed by the browser. We call this problem of *missing sanitization*. A significant body of prior research has focused on developing analysis for detecting program paths with missing sanitization.

In this chapter, we study why sanitization is challenging and determine how it is used in existing large-scale applications. In addition, we study the prominent defense techniques available to web developers in emerging web application frameworks. We find two new problems with how sanitization is used in large commercial applications, which go beyond the problems of missing sanitization or incorrect implementation of sanitizers. We also empirically measure the support for correct sanitization available to developers from 14 popular web frameworks. We begin by introducing the challenges in using a sanitization-based defense techniques correctly in the next section.

5.1 Challenges in Sanitization

Sanitization is deviously complex; it involves understanding how the web browser parses and interprets web content in non-trivial detail. Though immensely important, this issue has not been adequately explained in prior research. For instance, prior research does not detail the security ramifications of the complex interactions between the sub-languages implemented in the browser or the subtle variations in different interfaces for accessing or evaluating data via JavaScript's DOM API. This has important implications on the security of sanitization, as we show through multiple examples in this chapter. For instance, we show examples of how sanitization performed on the server-side can be effectively "undone" by the browser's parsing of content into the DOM, which may introduce scripting vulnerabilities in client-side JavaScript code.

We formulate the sanitization defense using a comprehensive model of the browser's parsing behavior in this section. We discuss the challenges and subtleties scripting sanitization must address here.

Sanitization Defined

Web applications mix control data (code) and content in their output which is consumed by the web browser. When data controlled by the attacker is interpreted by the web browser as if it was code written by the web developer, a scripting attack results. A canonical example of a scripting attack is as follows. Consider a blogging web application that emits untrusted content, such as anonymous comments, on the web page. If the developer is not careful, an attacker can input text such as `<script>...<script>`, which may be output verbatim in the server's output HTML page. When a user visits this blog page, her web browser will execute the attacker controlled text as script code.

Sanitization requires removal of such dangerous tags from the untrusted data. Unfortunately, not all cases are as simple as this `<script>` tag example. In the rest of this section, we identify browser features that make preventing script injection attacks much more complicated.

The Browser Model. We present a comprehensive model of the web browser's parsing behavior. While the intricacies of browser parsing behavior have been discussed before [136], a formal model has not been built to fully explore its complexity. We show this model in Figure 5.1. Abstractly, the browser can be viewed as a collection of HTML-related sub-grammars and a collection of transducers. Sub-grammars correspond to parsers for languages such as URI schemes, CSS, HTML, and JavaScript (the rounded rectangles in Figure 5.1). Transducers transform or change the representation of the text, such as in HTML-entity encoding/decoding, URI-encoding, JavaScript Unicode encoding and so on (the unshaded rectangles in Figure 5.1). The web application's output, i.e., HTML page, is input into the browser via the network; it can be directly fed into the HTML parser after some pre-processing or it can be fed into JavaScript's HTML evaluation constructs. The browser parses these input fragments in stages—when a fragment is recognized as a term in another

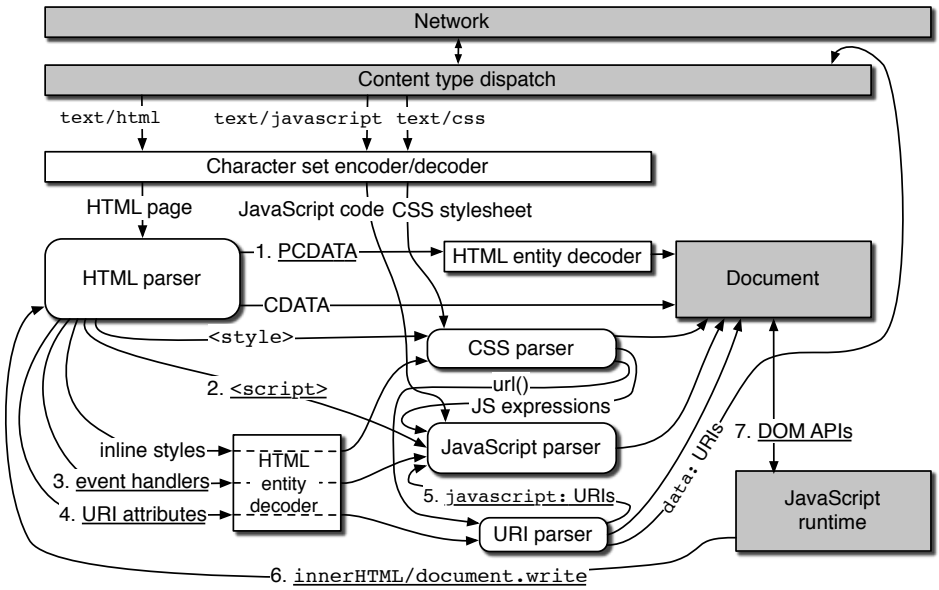

Figure 5.1: Flow of Data in our Browser Model. Certain contexts such as `PCDATA` and `CDATA` directly refer to parser states in the HTML 5 specification. We refer to the numbered and underlined edges during our discussion in the text.

sub-grammar, it is shipped to the corresponding sub-grammar for reparsing and evaluation (e.g., edge 2). For example, while the top-level HTML grammar identifies an anchor (`<a>`) tag in the HTML document, the contents of the `href` attribute are sent to the URI parser (edge 4). The URI parser handles a `javascript:` URI by sending its contents to the JavaScript parser (edge 3), while other URIs are sent to their respective parsers.

Subtleties and Challenges in Sanitization

The model shows that the interaction between sub-components is complex; burdening developers with fully understanding their subtleties is impractical. We now describe a number of such challenges that correct sanitization-based defense needs to address.

Challenge 1: Context Sensitivity. Sanitization for XSS defense requires knowledge of where untrusted input appears structurally and semantically in the web application. For example, simple HTML-entity encoding is a sufficient sanitization procedure to neutralize scripting attacks when is placed inside the body of an HTML tag, or, in the `PCDATA` (edge 1) parsing context, as defined by HTML5 [119]. However, when data is placed in a resource URI, such as the `src` or `href` attribute of a tag, HTML-encoding is insufficient to block attacks such as via a `javascript:` URI (edge 4 and 5). We term the intuitive notion of *where* untrusted data appears as its *context*. Sanitization requirements vary by contexts.

Frameworks providing sanitization primitives need to be mindful of such differences from context to context. The list of these differences is large [94].

Challenge 2: Sanitizing nested contexts. We can see in the model that a string in a web application's output can be parsed by multiple sub-parsers in the browser. We say that such a string is placed in *nested contexts*. That is, its interpretation in the browser will cause the browser to traverse more than one edge shown in Figure 5.1.

Sanitizing for nested contexts adds its own complexity. Consider an untrusted string embedded inside a script block, such as `<script> var x = ‘ UNTRUSTED DATA ’...</script>`. In this example, when the underlined data is read by the browser, it is simultaneously placed in two contexts. It is placed in a JavaScript string literal context by the JavaScript parser (edge 2) due to the single quotes. But, before that, it is inside a `<script>` HTML tag (or `RCDATA` context according to the HTML 5 specification) that is parsed by the HTML parser. Two distinct attack vectors can be used here: the attacker could use a single quote to break out of the JavaScript string context, or inject `</script>` to break out of the script tag. In fact, sanitizers commonly fail to account for the latter because they do not recognize the presence of nested contexts.

Challenge 3: Browser Transductions. If dealing with multiple contexts is not arduous enough, our model highlights the *implicit transductions* that browsers perform when handing data from one sub-parser to another. These are represented by edges from rounded rectangles to unshaded rectangles in Figure 5.1. Such transductions and browser-side modifications can, surprisingly, *undo* sanitization applied on the server.

Consider a blog page in which comments are hidden by default and displayed only after a user clicks a button. The code uses an `onclick` JavaScript event handler:

```
<div class=‘comment-box’onclick=‘displayComment(" UNTRUSTED ",this)’>
   ... hidden comment ... </div>
```

The underlined untrusted comment is in two nested contexts: the HTML attribute and single-quoted JavaScript string contexts. Apart from preventing the data from escaping out of the two contexts separately (Challenge 2), the sanitization must worry about an additional problem. The HTML 5 standard mandates that the browser HTML-entity decode an attribute value (edge 3) before sending it to a sub-grammar. As a result, the attacker can use additional attack characters even if the sanitization performs HTML-entity encoding to prevent attacks. The characters `&quot;` will get converted to `"` before being sent to the JavaScript parser. This will allow the untrusted comment to break out of the string context in the JavaScript parser. We term such implicit conversions as *browser transductions*.

Table 5.1 details browser transductions that are automatically performed upon reading or writing to the DOM. The DOM property denotes the various aspects of an element accessible through the DOM APIs, while the access method describes the specific part of the API through which a developer may edit or examine these attributes. Excepting “specified in markup”, the methods are all fields or functions of DOM elements.

Table 5.2 describes the specifics of the transducers employed by the browser. Except for “HTML entity decoding”, the transductions all occur in the parsing and serialization

DOM property	Access method	Transductions on reading	Transductions on writing
`data-*` attribute	`get/setAttribute`	None	None
	`.dataset`	None	None
	specified in markup	N/A	HTML entity decoding
src, href attributes	`get/setAttribute`	None	None
	`.src`, `.href`	URI normalization	None
	specified in markup	N/A	HTML entity decoding
id, alt, title, type, lang, class dir attributes	`get/setAttribute`	None	None
	.[attribute name]	None	None
	specified in markup	N/A	HTML entity decoding
style attribute	`get/setAttribute`	None	None
	`.style.*`	CSS serialization	CSS parsing
	specified in markup	N/A	HTML entity decoding
HTML contained by node	`.innerHTML`	HTML serialization	HTML parsing
Text contained by node	`.innerText`, `.textContent`	None	None
HTML contained by node, including the node itself	`.outerHTML`	HTML serialization	HTML parsing
Text contained by node, surrounded by markup for node	`.outerText`	None	None

Table 5.1: Transductions applied by the browser for various accesses to the document. These summarize transductions when traversing edges connected to the "Document" block in Figure 5.1.

Type	Description	Illustration
HTML entity decoding	Replacement of character entity references with the actual characters they represent.	`&` → `&`
HTML parsing	Tokenization and DOM construction following the HTML parsing rules, including entity decoding as appropriate.	`<p>&gt;</p>` → *HTML element* P *with body* `>`
HTML serialization	Creating a string representation of an HTML node and its children.	*HTML element* P *with body* `>` → `<p>&gt;</p>`
URI normalization	Resolving the URI to an absolute one, given the context in which it appears.	`/article title` → `http://www.example.com/article%20title`
CSS parsing	Parsing CSS declarations, including character escape decoding as appropriate.	`color: \72\65\64` → `color: red`
CSS serialization	Creating a canonical string representation of a CSS style declaration.	" `color:#f00`" → " `color: rgb(255, 0, 0);` "

Table 5.2: Details regarding the transducers mentioned in Table 5.1. They all involve various parsers and serializers present in the browser for HTML and its related sub-grammars.

processes triggered by reading and writing these properties as strings. When writing to a property, the browser parses the string to create an internal AST representation. When reading from a property, the browser recovers a string representation from the AST.

Textual values are HTML entity decoded when written from the HTML parser to the DOM via edge 1 in Figure 5.1. Thus, when a program reads a value via JavaScript, the value is entity decoded. In some cases, the program must re-apply the sanitization to this decoded value or risk having the server's sanitization be undone.

One set of DOM read access APIs creates a serialized string of the AST representation of an element, as described in Table 5.2. The other API methods simply read the text values of the string versions (without serializing the ASTs to a string) and perform no canonicalization of the values.

The transductions vary significantly for the DOM write access API as well, as detailed in Table 5.1. Some writes cause input strings to be parsed into an internal AST representation, or apply simple replacements on certain character sequences (such as URI percent-decoding), while others store the input as is.

In addition, the parsers in Figure 5.1 apply their own transductions internally on certain pieces of their input. The CSS and JavaScript parsers unescape certain character sequences within string literals (such as Unicode escapes), and the URI parser applies some of its own as well (undoing percent-encoding).

Challenge 4: Dynamic Code Evaluation. In principle, the chain of edges traversed by the browser while parsing a text can be arbitrarily long because the browser can dynamically evaluate code. Untrusted content can keep cycling through HTML and JavaScript contexts. For example, consider the following JavaScript code fragment:

```
function foo(untrusted) {
    document.write("<input onclick='foo(" + untrusted + ")' >");
}
```

Since `untrusted` text is repeatedly pumped through the JavaScript string and HTML contexts (edges 3 and 6 of Figure 5.1), statically determining the context traversal chain on the server is infeasible. In principle, purely server-side sanitization is not sufficient for context determination because of dynamic code evaluation. Client-side sanitization is needed in these cases to fully mitigate potential attacks. Failure to properly sanitize such dynamic evaluation leads to the general class of attacks called DOM-based scripting or client-side code injection attacks [113].

Another key observation is that browser transductions along the edges of Figure 5.1 vary from one edge to another, as detailed earlier. This mismatch can cause scripting vulnerabilities. To illustrate this, we present a real-world example in Figure 5.2 from one of the applications we evaluated, `phpBB3`, showing how these subtleties may be misunderstood by developers.

In the server-side code, which is not shown here, the application sanitizes the `title` attribute of an HTML element by HTML-entity encoding it. If the attacker enters a string

```
text = element.getAttribute('title');
// ... elided ...
desc = create_element('span', 'bottom');
desc.innerHTML = text;
tooltip.appendChild(desc);
```

Figure 5.2: A real-world vulnerability in PHPBB3.

like `<script>`, the encoding converts it to `&lt;script&gt;`. The client-side code subsequently reads this attribute via the `getAttribute` DOM API in JavaScript code (shown above) and inserts it back into the DOM via the `innerHTML` method. The vulnerability is that the browser automatically decodes HTML entities (through edge 1 in Figure 5.1) while constructing the DOM. This effectively undoes the server's sanitization in this example. The `getAttribute` DOM API reads the decoded string (e.g., `<script>`) from the DOM (edge 7). Writing `<script>` via `innerHTML` (edge 6) results in XSS.

This bug is subtle. Had the developer used `innerText` instead of `innerHTML` to write the data, or used `innerHTML` to read the data, the code would *not* be vulnerable. The reason is that the two DOM APIs discussed here read different serializations of the parsed page, as explained earlier in this section.

Challenge 5: Character-set Issues. Successfully sanitizing a string at the server side implicitly requires that the sanitizer and the browser employ the same character set while working with the string. A common source of scripting vulnerabilities is a mismatch in the charset assumed by the sanitizer and the charset used by the browser. For example, the ASCII string `+ADw-` does not have any suspicious characters. But when interpreted by the browser as UTF-7 character-set, it maps to the dangerous `<` character: this mismatch between the server-side sanitization and browser character set selection has led to multiple scripting vulnerabilities [115].

Challenge 6: MIME-based XSS, Universal XSS, and Mashup Confinement. Browser quirks, especially in interpreting content or MIME types [11], contribute their own share of XSS vulnerabilities. Similarly, bugs in browser implementations, such as capability leaks [35] and parsing inconsistencies [10], or in browser extensions [14] are important components of the XSS landscape. However, these do not pertain to sanitization defenses in web frameworks. Therefore, we consider them to be out-of-scope for this study.

5.2 Support for Auto-Sanitization in Existing Web Application Frameworks

Though defense techniques for scripting attacks have enjoyed intense focus [16, 93, 110, 67, 118, 6, 132, 71, 127, 62, 55, 87, 75, 8, 15], research has paid little attention to a promising sets of tools—*web application frameworks*—which are gaining wide adoption [46, 27, 30,

103, 95, 32, 137, 133, 60, 112, 109]. Many of these frameworks claim that their sanitization abstractions can be used to make web applications secure against scripting attacks [111, 133]. Though possible in principle, this section investigates the extent to which it is presently true, clarifies the assumptions that frameworks make, and outlines the fundamental challenges that frameworks need to address.

Most legacy web applications implement their sanitization defense manually, which is prone to errors. Web frameworks offer a platform to automate sanitization in web applications, freeing developers from existing ad-hoc and error-prone manual analysis. As web applications increasingly rely on web frameworks, we aim to understand the assumptions web frameworks build on and the security of their underlying sanitization mechanisms.

A web framework can address scripting attacks using sanitization if it correctly addresses all the subtleties. Whether existing frameworks achieve this goal is an important question and a subject of this chapter. A systematic study of today's web frameworks should evaluate their security and assumptions along the following dimensions to quantify their benefits:

- **Context Expressiveness and Sanitizer Correctness.** As we detailed in Challenge 1, sanitization requirements change based on the context of the untrusted data. We investigate the set of contexts in which untrusted data is used by applications, and whether web frameworks support those contexts. In the absence of such support, a developer will have to revert to manually writing sanitization functions. The challenges outlined in Section 5.1 make manually developing *correct* sanitizers a non-starter. Instead, we ask, *do web frameworks provide correct sanitizers for different contexts that applications commonly use in practice?*

- **Auto-sanitization and Context-Sensitivity.** Providing sanitizers is only a small part of the overall solution necessary to defend against scripting attacks attacks. Applying sanitizers in code automatically, which we term *auto-sanitization*, shifts the burden of ensuring safety against scripting attacks from developers to frameworks. The benefit of this is self-evident: performing correct sanitization in framework code spares each and every developer from having to implement correct sanitization himself, and from having to remember to perform that sanitization everywhere necessary. Furthermore, correct auto-sanitization needs to be context-sensitive—context-insensitive auto-sanitization can lead to a false sense of security. *Do web frameworks offer auto-sanitization, and if so, is it context-sensitive?*

In this section, we empirically analyze web frameworks and the sanitization abstractions they provide along the outlined dimensions. We compare application requirements to each abstraction provided by frameworks, showing that there is a mismatch in the abstractions provided by frameworks and the requirements of applications.

We begin by analyzing the "auto-sanitization" feature—a security primitive in which web frameworks sanitize untrusted data automatically—in Section 5.2. We identify the extent to which it is available, the pitfalls of its implementation, and whether developers can

Language	Framework, Plugin, or Feature	Automatically Sanitizes in HTML Context	Performs Context-Aware Sanitization	Pointcut
PHP	CodeIgniter	•		Request Reception
VB, C#, C++, F#	ASP.NET Request Validation [80]	•		Request Reception
Ruby	xss_terminate Rails plugin [131]	•		Database Insertion
Python	Django	•		Template Processing
Java	GWT SafeHtml	•	•	Template Processing
C++	Ctemplate	•	•	Template Processing
Language-neutral	ClearSilver	•	•	Template Processing

Table 5.3: Extent of automatic sanitization support in the frameworks we study and the pointcut (set of points in the control flow) where the automatic sanitization is applied.

blindly trust this mechanism if they migrate/develop applications on existing auto-sanitizing frameworks.

Frameworks may not provide auto-sanitization, but instead may provide sanitizers which developers can manually invoke. Arguably, the sanitizers implemented by frameworks would be more robust than the ones implemented by the application developer. We evaluate the breadth of contexts for which each framework provides sanitizers, or the *context expressiveness* of each framework, later in this section. We also compare it to the requirements of the applications we study today to check if this expressiveness is enough for real-world applications.

Methodology and Analysis Subjects. We examine 14 popular web application frameworks in commercial use for different programming languages and 8 popular PHP web applications ranging from 19 KLOC to 532 KLOC in size. We used a mixture of manual and automated exploration to identify sanitizers in the web application running on an instrumented PHP interpreter. We then executed the application again along paths that use these sanitization functions and parse the outputs using a HTML 5-compliant browser to determine the contexts for which they sanitize. We focus here solely on the results of our empirical analysis. A technical report provides the full details of the techniques employed [126].

Auto-Sanitization: Features and Pitfalls

Auto-sanitization is a feature that shifts the burden of ensuring safety against scripting from the developer to the framework. In a framework that includes auto-sanitization, the application developer is responsible for indicating which variables will require sanitization. When the page is output, the web application framework can then apply the correct sanitizer to these variables. Our findings, summarized in Table 5.3, are as follows:

- Of the 14 frameworks evaluated, only 7 support some form of auto-sanitization.
- 4 out of the 7 auto-sanitization framework apply a "one-size-fits-all" strategy to sanitization. That is, they apply the same sanitizer to all flows of untrusted data irrespective

of the context into which the data flows. We call this *context-insensitive* sanitization, which is fundamentally unsafe, as explained later.

- We measure the fraction of application output sinks actually protected by context-insensitive auto-sanitization mechanism in 10 applications built on Django, a popular web framework. Table 5.4 presents our findings. The mechanism fails to correctly protect between 14.8% and 33.6% of an application's output sinks.
- Only 3 frameworks perform context-sensitive sanitization.

No auto-sanitization. Only half of the studied frameworks provide any auto-sanitization support. This implies that developers must deal with the challenges of selecting where to apply built-in or custom sanitizers in code when using such frameworks. This manual process is prone to errors, as evidenced in the following real-world example from a Django application called GRAMPS.

```
{% if header.sortable %}
    <a href="{{header.url|escape}}">
{% endif %}
```

Figure 5.3: Example of Django application with wrong sanitization

The developer sanitizes a data variable placed in the `href` attribute but uses the HTML-entity encoder (`escape`) to sanitize the data variable `header.url`. This is an instance of Challenge 2 outlined in Section 5.1. In particular, this sanitizer fails to prevent scripting attack vectors such as `javascript:` URIs.

Insecurity of Context-insensitive auto-sanitization. Another interesting fact about the above example is that even if the developer relied on Django's default auto-sanitization, the code would be vulnerable to scripting attacks. Django employs context-insensitive auto-sanitization, i.e., it applies the same sanitizer (`escape`) irrespective of the output context. The sanitization primitive `escape` performs an HTML-entity encode and is thus safe for use in HTML tag context but unsafe for other contexts. In the above example, applying `escape`, automatically or otherwise, fails to protect against scripting attacks. Auto-sanitization support in Rails [131], .NET (request validition [80]) and CodeIgniter is context-insenstive and has similar problems.

Context-insensitive auto-sanitization provides a false sense of security. On the other hand, relying on developers to pick a sanitizer consistent with the context is error-prone, and one scripting hole is sufficient to subvert the web application's integrity. Thus, because it covers some limited cases, context-insensitive auto-sanitization is better protection than no auto-sanitization.

We measure the percentage of output sinks protected by context-insensitive auto-sanitization in 10 Django-based applications that we randomly selected for further investigation [33]. We

Web Application	No. Sinks	% Auto-sanitized Sinks	% Sinks not sanitized (marked safe)	% Sinks manually sanitized	% Sinks in HTML Context	% Sinks in URI Attr. (excl. scheme)	% Sinks in URI Attr. (incl. scheme)	% Sinks in JS Attr. Context	% Sinks in JS Number or String Context	% Sinks in Style Attr. Context
GRAMPS Genealogy Management	286	77.9	0.0	22.0	66.4	3.4	30.0	0.0	0.0	0.0
HicroKee's Blog	92	83.6	7.6	8.6	83.6	6.5	7.6	1.0	0.0	1.0
FabioSouto.eu	55	90.9	9.0	0.0	67.2	7.2	23.6	0.0	1.8	0.0
Phillip Jones' Eportfolio	94	92.5	7.4	0.0	73.4	11.7	12.7	0.0	2.1	0.0
EAG cms	19	94.7	5.2	0.0	84.2	0.0	5.2	0.0	0.0	10.5
Boycott Toolkit	347	96.2	3.4	0.2	71.7	1.1	25.3	0.0	1.7	0.0
Damned Lies	359	96.6	3.3	0.0	74.6	0.5	17.8	0.0	0.2	6.6
oebfare	149	97.3	2.6	0.0	85.2	6.0	8.0	0.0	0.0	0.6
Malaysia Crime	235	98.7	1.2	0.0	77.8	0.0	1.7	0.0	20.4	0.0
Philippe Marichal's web site	13	100.0	0.0	0.0	84.6	0.0	15.3	0.0	0.0	0.0

Table 5.4: Usage of auto-sanitization in Django applications. The first 2 columns are the number of sinks in the templates and the percentage of these sinks for which auto-sanitization has not been disabled. Each remaining column shows the percentage of sinks that appear in the given context.

statically correlated the automatically applied sanitizer to the context of the data; the results are in Table 5.4. The mechanism protects between 66.4% and 85.2% of the output sinks, but conversely permits scripting vectors in 14.8% to 33.6% of the contexts, subject to whether attackers control the sanitized data or not. We did not determine the exploitability of these incorrectly auto-sanitized cases, but we observed that in most of these cases, developers resorted to custom manual sanitization.

Context-Sensitive Sanitization. Context-sensitive auto-sanitization addresses the above issues. Three web frameworks, namely GWT, Google Clearsilver, and Google Ctemplate, provide this capability. In these frameworks, the auto-sanitization engine performs runtime parsing, keeping track of the context before emitting untrusted data. The correct sanitizer is then automatically applied to untrusted data based on the tracked context. These frameworks rely on developers to identify untrusted data. The typical strategy is to have developers write code in *templates*, which separate the HTML content from the (untrusted) data variables. For example, consider the following simple template supported by the Google

Ctemplate framework shown in Figure 5.4.

```
{{%AUTOESCAPE context="HTML"}}
<html><body><script> function showName() {
document.getElementById("sp1").textContent = "Name: {{NAME}}";} </script>
<span id="sp1" onclick="showName()">Click to display name.</span><br/>
Homepage: <a href="{{URI}}"> {{PAGENAME}} </a></body></html>
```

Figure 5.4: Example of Auto-sanitization in Google Ctemplate framework

Variables that require sanitization are surrounded by `{{` and `}}`; the rest of the text is HTML content to be output. When the template executes, the engine parses the output and determines that (for instance) `{{NAME}}` is in a JavaScript string context and automatically applies the sanitizer for the JavaScript string context, namely `:javascript_escape`. For other variables, the same mechanism applies the appropriate sanitizers. For instance, the variable `{{URI}}` is sanitized with the `:url_escape_with_arg=html` sanitizer.

Context Expressiveness

Having analyzed the auto-sanitization support in web frameworks for static HTML evaluation as well as dynamic evaluation via JavaScript, we turn to the support for manual sanitization. Frameworks may not provide auto-sanitization but instead may provide sanitizers which developers can call. This improves security by freeing the developer from (re)writing complex, error-prone sanitization code. In this section, we evaluate the breadth of contexts for which each framework provides sanitizers, or the *context expressiveness* of each framework. For example, a framework that provides built-in sanitizers for more than one context, say in URI attributes, CSS keywords, JavaScript string contexts, is more expressive than one that provides a sanitizer only for HTML tag context.

Expressiveness of Framework Sanitization Contexts. Table 5.5 presents the expressiveness of web frameworks we study and Table 5.6 presents the expressiveness required by our subject web applications. The key insights are:

- We observe that 9 out of the 14 frameworks do not support contexts other than the HTML context (e.g., as the content body of a tag or inside a non-URI attribute) and the URI attribute context. The most common sanitizers for these are HTML entity encoding and URI encoding, respectively.

- 4 web frameworks, ClearSilver, Ctemplate, Django, and Smarty, provide appropriate sanitization functions for emitting untrusted data into a JavaScript string. Only 1 framework, Ctemplate, provides a sanitizer for emitting data into JavaScript outside of the string literal context. However, the sanitizer is a restrictive whitelist, allowing only

Language	Framework	HTML tag content or non-URI attribute	URI Attribute (excluding scheme)	URI Attribute (including scheme)	JS String	JS Number or Boolean	Style Attribute or Tag
Perl	Mason [1, 112]	•	•				
	Template Toolkit [109]	•	•				
	Jifty [60]	•	•				
PHP	CakePHP [23]	•	•				
	Smarty Template Engine [103]	•	•		•		
	Yii [133, 53]	•	•				
	Zend Framework [137]	•	•				
	CodeIgniter [28, 29]	•	•				
VB, C#, C++, F#	ASP.NET [52]	•	•				
Ruby	Rails [95]	•	•				
Python	Django [32]	•	•	•	•		
Java	GWT SafeHtml [46]	•	•	•			
C++	Ctemplate [30]	•	•	•	•	•	•
Language-neutral	ClearSilver [27]	•	•	•	•		•

Table 5.5: Sanitizers provided by languages and/or frameworks. For frameworks, we also include sanitizers provided by standard packages or modules for the language.

numeric or boolean literals. No framework we studied allows untrusted JavaScript code to be emitted into JavaScript contexts. Supporting this requires a client-side isolation mechanism such as ADsafe [3] or Google's Caja [45].

- 4 web frameworks, namely Django, GWT, Ctemplate, and Clearsilver, provide sanitizers for URI attributes in which a complete URI (i.e., including the URI protocol scheme) can be emitted. These sanitizers reject URIs that use the `javascript:` scheme and accept only a whitelist of schemes, such as `http:`.

- Of the frameworks we studied, we found only one that provides an interface for customizing the sanitizer for a given context. Yii uses HTML Purifier [53], which allows the developer to specify a custom list of allowed tags. For example, a developer may specify a policy that allows only `<b>` tags. The other frameworks (even the context-sensitive auto-sanitizing ones) have sanitizers that are not customizable. That is, untrusted content within a particular context is always sanitized the same way.

The set of contexts for which a framework provides sanitizers gives a sense of how the framework expects web applications to behave. Specifically, frameworks assume applications will not emit sanitized content into multiple contexts. More than half of the frameworks we examined do not expect web applications to insert content with arbitrary schemes into URI contexts, and only one of the frameworks supports use of untrusted content in JavaScript

Application	Description	LOC	HTML Context	URI Attr. (excl. scheme)	URI Attr. (incl. scheme)	JS Attr. Context	JS Number or String Context	No. of Sanitizers	No. of Sinks
RoundCube	IMAP Email Client	19,038	•	•	•	•	•	30	75
Drupal	Content Management	20,995	•	•	•	•	•	32	2557
Joomla	Content Management	75,785	•	•	•	•		22	538
WordPress	Blogging App.	89,504	•	•	•	•		95	2572
MediaWiki	Wiki Hosting	125,608	•	•	•	•	•	118	352
PHPBB3	Bulletin Board Software	146,991	•	•	•	•	•	19	265
OpenEMR	Medical Records Mgmt.	150,384	•			•	•	18	727
Moodle	E-Learning Software	532,359	•	•	•	•	•	43	6282

Table 5.6: The web applications we study and the contexts for which they sanitize.

`Number` or `Boolean` contexts. Below, we challenge these assumptions by quantifying the set of contexts for which applications need sanitizers.

Expressiveness of Contexts in Web Applications. We examined our 8 subject PHP applications, ranging from 19 to 532 KLOC, to understand what expressiveness they require and whether they could, theoretically, migrate to the existing frameworks. We systematically measure and enumerate the contexts into which these applications emit untrusted data. Table 5.6 shows the result of this evaluation. We observe that nearly all of the applications insert untrusted content into all of the outlined contexts. Contrast this with Table 5.5, where most frameworks support a much more limited set of contexts with built-in sanitizers.

5.3 Failures of Sanitization in Large-Scale Applications

In this section, we present our analysis of 7 large-scale, Microsoft .NET-based shipping web applications. These applications have a total of over 400,000 lines of code. We performed our security testing on a set of 53 large web pages derived from these 7 applications. Each page contains 350–900 DOM nodes. We have found two new class of sanitization errors we commonly observe in our empirical analysis: *context-mismatched sanitization* and *inconsistent multiple sanitization*, both of which demonstrate that placement of sanitizers in legacy code is a significant challenge even if the sanitizers themselves are securely constructed. As mentioned previously, these errors are not the traditional known problems of missing sanitization or those of incorrectly implemented sanitizers; each individual sanitizer in our test application is *correct* to the best of our knowledge. We explain these errors below and give an overview of our empirical findings. We exclude the implementation and design of a tool called ScriptGard which we developed to perform this analysis—details of this tool are available in our conference paper [97].

HTML Sink Context	Correct sanitizer that suffices
HTML Tag Context	`HTMLEncode`, `SimpleTextFormatting`
Double Quoted Attribute	`HTMLAttribEncode`
Single Quoted Attribute	`HTMLAttribEncode`
URL Path attribute	`URLPathEncode`
URL Key-Value Pair	`URLKeyValueEncode`
In Script String	`EcmaScriptStringEncode`
CDATA	`HTMLEncode`
Style	`Alpha − numerics`

Figure 5.5: Sanitizer-to-context mapping for our test applications.

Scripting attacks are highly *context*-dependent—a string such as `expression: alert(a)` is innocuous when placed inside a HTML tag context, but can result in JavaScript execution when embedded in a CSS attribute value context. In fact, the set of contexts in which untrusted data is commonly embedded by today's web applications is well-known. Sanitizers that secure data against scripting attacks in each of these contexts are publicly available [129, 42]. We find that security experts have already implemented functionally correct sanitizers and specified the contexts that each sanitizer corresponds to for our target application. We extract this security specification by discussing with the security experts and the mapping of sanitizers to their intended contexts of use are shown in Figure 5.5. We refer to this mapping in the rest of this chapter.

Inconsistent Multiple Sanitization

We illustrate the new classes of errors with an example. Figure 5.6 shows a fragment of ASP.NET code written in C# which illustrates the difficulty of sanitizer placement. This running example is inspired by code from the large code base we empirically analyzed. Consider the function `DynamicLink.RenderControl` shown in the running example, which places an untrusted string inside a double-quoted `href` attribute which in turn is placed inside a JavaScript string. This code fragment places the untrusted string into two nested contexts—when the browser parses the untrusted string, it will first parse it in the JavaScript string literal and then subsequently parse it as a URI attribute (as explained in Section 5.1).

In Figure 5.5 we show a sanitizer specification that maps functions to contexts. In particular, two sanitizer functions, `EcmaScriptStringEncode` and `HtmlAttribEncode`, are applied for the JavaScript string context and the HTML attribute context, respectively. However, developers must understand this order of parsing in the browser to apply them in the correct order. They must choose between the two ways of composing the two sanitizers shown in Figure 5.7.

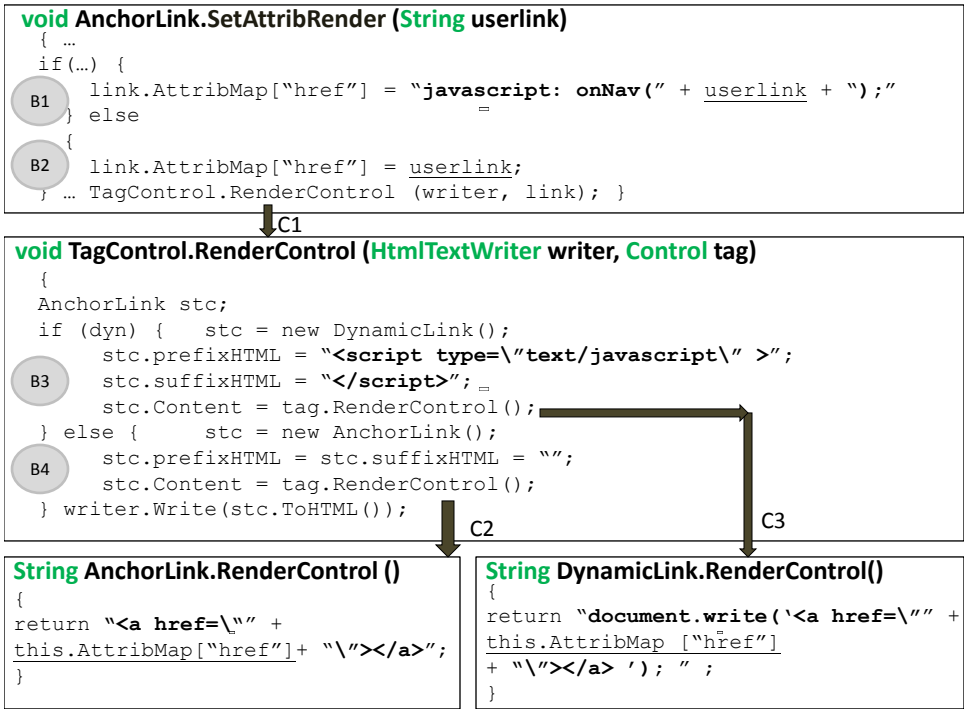

Figure 5.6: Running example: C# code fragment illustrating the problem of automatic sanitizer placement. Underlined values are derived from untrusted data and require sanitization; function calls are shown with thick black arrows C1-C3 and basic blocks B1-B4 are shown in gray circles.

It turns out that the first sequence of sanitizer composition is *inconsistent* with the nested contexts, while the other order is safe or consistent. We observe that the standard recommended implementation for these sanitizers [129] do not commute. For instance, `EcmaScriptStringEncode` simply transforms all characters that can break out of JavaScript string literals (like the `"` character) to Unicode encoding (`\u0022` for `"`), and, `HtmlAttribEncode` HTML-entity encodes characters (`&quot;` for `"`). This is the standard recommended behavior these sanitizers [129] with respect to respective contexts they secure.

```
document.write('<a href=" +  HtmlAttribEncode(EcmaScriptStringEncode(this.AttribMap["href"]))
document.write('<a href=" + EcmaScriptStringEncode(HtmlAttribEncode(this.AttribMap["href"]))
```

Figure 5.7: Two different sanitization approaches are shown: Method 1 is shown above and method 2 below.

The attack on the wrong composition is subtle. The key observation is that applying `EcmaScriptStringEncode` first encodes the attacker-supplied `"` character as a Unicode representation `\u0022`. This Unicode representation is not subsequently transformed by the second

`HtmlAttribEncode` sanitization, because `\u0022` is a completely innocous string in the URI attribute value context.

However, when the web browser parses the transformed string first (in the JavaScript string literal context), it performs a Unicode decode of the dangerous character `\u0022` back to `"`. When the browser subsequently interprets this in the `href` URI attribute context the attacker's dangerous `"` prematurely closes the URI attribute value and can inject JavaScript event handlers like `onclick=...` to execute malicious code. It is easy to confirm that the other composition of correct sanitizers is definitely safe.

Context-Mismatched Sanitization

Even if sanitizers are designed to be commutative, developers may apply a sanitizer that does not match the context altogether; we call such an error as a *context-mismatched sanitization* inconsistency. Context-mismatched sanitization is not uncommon in real applications. To intuitively understand why, consider the sanitization requirements of the running example again.

Notice that the running example has 4 control-flow paths corresponding to execution through the basic-blocks (B1,B3), (B1,B4), (B2,B3) and (B2,B4) respectively. Each execution path places the untrusted `userlink` input string in 4 different contexts (see Figure 5.8). Determining the browser context is a path-sensitive property, and the developer may have to inspect the global control/data flow to understand in which contexts is a data variable used. This task can be error-prone because the code logic for web output may be spread across several classes, and the control-flow graph may not be explicit (especially in languages with inheritance). We show how two most prevalent sanitization strategies fail to work on this example.

Failure of output sanitization. Consider the case in which the application developer decides to delay all sanitization to the *output* sinks, i.e., to the `writer.Write` call in `TagControl.RenderControl`. There are two problems with doing so, which the developer is burdened to identify manually to get the sanitization right. First, the execution paths through basic-block B3 embed the untrusted data in a `<SCRIPT>` block context, where paths through basic-block B4 place it in a HTML tag context. As a result, *any* sanitizer picked cannot be consistent for both such paths. Second, even if the first concern did not exist, sanitizing the `stc.Content` variable at the output point is *not* correct. The `stc.Content` is composed of trusted substrings as well as untrusted data — if the entire string is sanitized, the sanitizer could change programmer-supplied constant strings in a way that breaks the intended structure of the output HTML. For example, if the basic-block B1 executes, the untrusted data would be embedded in a JavaScript number context(`javascript: OnNav()` explicitly by the programmer. If we applied `HtmlAttribEncode` to the `stc.Content` the `javascript:` would be eliminated breaking the application's intended behavior.

Failure of input sanitization. Moving sanitization checks to earlier points in the code, say at the input interfaces, is not a panacea either. The readers can verify that moving all

HTML output	Nesting of contexts

`<script type="text/javascript">`
`  document.write(‘`
`  <a href="javascript: onNav(`
`TOENCODE )";></a>);`
`</script>`
	JavaScript String Literal,
	Html URI Attribute,
	JavaScript Number

`<a href="javascript: onNav(` `TOENCODE );"></a>`	Html URI Attribute, JavaScript Number
`<a href="TOENCODE"></a>`	Html URI Attribute

`<script type="text/javascript">`
`  document.write(‘ <a href="`
`TOENCODE"> </a>’);`
`</script>`
	JavaScript String Literal,
	Html URI Attribute

Figure 5.8: HTML outputs obtained by executing different paths in the running example. TOENCODE denotes the untrusted string in the output.

sanitization to a code locations earlier in the dataflow graph continues to suffer from path-sensitivity issues. Sanitizing in basic-blocks B1 and B2 is not sufficient, because additional contexts are introduced when blocks B3 and B4 are executed. Sanitization locations midway in the dataflow chain, such the concatenation in function `AnchorLink.SetAttribRender`, are also problematic because depending on whether basic-block B1 executes or B2 executes, the `this.AttribMap["href"]` variable may have trusted content or not.

In the next section, we empirically analyze the extent to which these inconsistency errors arise in practical real-world code we study. We point out that state-of-the-art static analysis tools which scale to hundred-thousand LOC applications, presently are fairly limited. Most existing tools detect data-flow paths with sanitizers missing altogether. This class of errors is detected by several static or dynamic analysis tools (such as CAT.NET [81] or Fortify [37]). Our study finds errors where sanitization is present but is inconsistent. Our evaluation studies a large legacy application of over 400,000 lines of server-side C# code. We accessed 53 distinct web pages by manually executing this application using our testing infrastructure [97]. Our analysis statically instrumented 23,244 functions.

Figure 5.5 shows the mapping between contexts and sanitization functions for our application. In particular, it permits only quoted attributes which have well-defined rules for sanitization [129]. Furthermore, it always sets the page encoding to UTF-8, eliminating the possibility of character-set encoding attacks [101]. We arrived at the result in Figure 5.5 after several interactions with the application's security engineers.

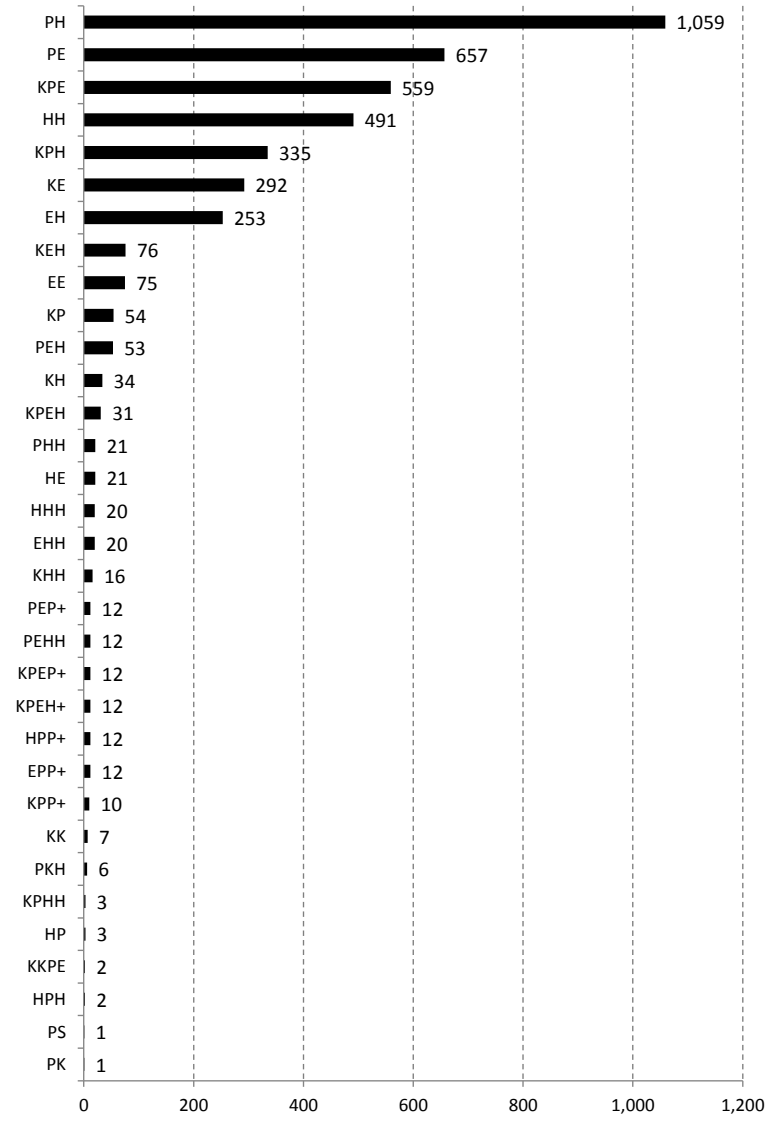

Figure 5.9: Histogram of sanitizer sequences consisting of 2 or more sanitizers empirically observed in analysis, characterizing sanitization practices resulting from manual sanitizer placement. `E`,`H`,`U`, `K`,`P`,`S` denote sanitizers `EcmaScriptStringLiteralEncode`, `HtmlEncode`, `HtmlAttribEncode`, `UrlKeyValueEncode`, `UrlPathEncode`, and SimpleHtmlEncode respectively.

Analysis Results

We report the number of context-mismatched and inconsistent multiple sanitization cases we find in our evaluation.

Context-mismatched sanitization. Figure 5.10 shows that our testing exercised 25,209 paths on which sanitization was applied[1]. Of these, 1,558 paths (or 6.1%) were improperly sanitized. Of these improperly sanitized paths, 1,207 (4.7% of the total analyzed paths) contained data that could not be proven safe by our testing infrastructure; as a result, we mark them as context inconsistency errors and revert to additional mechanisms to nullify these errors at runtime [97]. The difference between last and second last column in Figure 5.10 is the paths that sanitize constant strings or provably trusted data.

We used Red Gate's .NET Reflector tool, combined with other decompilation tools, to further investigate the executions which our testing tool reported as improperly sanitized. Our subsequent investigation reveals that errors result because it is difficult to manually analyze the calling context in which a particular portion of code may be invoked. In particular, the source and the sink may be separated by several intervening functions. Since our testing infrastructure instruments all string operations, we can count how far sources and sinks are removed from each other. In Figure 5.11, we graph the distribution of these lengths for a randomly selected sample of untrusted paths. This shows that a significant fraction of the chains are long and over 200 of them exceed 5 steps.

Our data on the length of def-use chains is consistent with those reported in previous static analysis based work [71]. As explained earlier in this Section, the sharing of dataflow paths can result in further ambiguity in distinguishing context at the HTML output point in the server, as well as, in distinguishing trusted data from untrusted data. In our investigation we observed the following cases:

- A single sanitizer was applied in a context that did not match. Typically, the sanitizer applied was in a different function from the one that constructed the HTML template. This suggests that developers may not fully understand how the context — a global property — impacts the choice of sanitizer, which is a local property. This is not surprising, given the complexity of choices in Figure 5.5.
- A sanitizer was applied to trusted data (on 1.4% of the paths in our experiment). We still report these cases because they point to developer confusion. On further investigation, we determined this was because sinks corresponding to these executions were shared by several dataflow paths. Each such sink node could output potentially untrusted data on some executions, while outputting purely trusted data on others.
- More than one sanitizer was applied, but the applied sanitizers were not correct for the browser parsing context of the data[2].

Inconsistent Multiple Sanitization. We found 3,245 paths with more than one sanitizer. Of these, 285 (or 8%) of the paths with multiple sanitization were inconsistent with the context. The inconsistent paths fell into two categories: first, we found 273 instances

[1]Each path here refers to a dataflow path, which starts at a program point where an untrusted input enters the application and ends in a critical operation which writes HTML to the output stream

[2]Errors where the combination was correct but the ordering was inconsistent with the nested context are reported separately as inconsistent multiple sanitization errors.

with the (`EcmaScriptStringLiteralEncode`)(`HtmlEncode`)+ pattern applied. As we saw in Section 5.3, these sanitizers do not commute, and this specific order is inconsistent. Second, we found 12 instances of the (`EcmaScriptStringLiteralEncode`)(`UrlPathEncode`)+ pattern. This pattern is inconsistent because it does not properly handle sanitization of URL parameters. If an adversary controls the data sanitized, it may be able to inject additional parameters.

We found an additional 498 instances of multiple sanitization that were superfluous. That is, sanitizer A was applied before sanitizer B, rendering sanitizer B superfluous. While not a security bug, this multiple sanitization could break the intended functionality of the applications. For example, repeated use of `UrlKeyValueEncode` could lead to multiple percent encoding causing broken URLs. Repeated use of `HtmlEncode` could lead to multiple HTML entity-encoding causing incorrect rendering of output HTML.

We also observed that nesting of parsing contexts is common. For example a URL may be nested within an HTML attribute. Figure 5.3 shows the histogram of sanitizer sequence lengths observed. The inferred context for a majority of these sinks demanded the use of multiple sanitizers. Figure 5.9 shows the use of multiple sanitizers in the application is widespread, with sanitizer sequences such as `UrlPathEncode HtmlEncode` being most popular. In our application, these sanitizers are not commutative, i.e. they produce different outputs if composed in different orders, which means that paths with different orderings produce different behavior.

Because our testing uses a dynamic technique [97], all paths found can be reproduced with test cases exhibiting the context-inconsistent sanitization. We investigated a small fraction of these test cases in more depth. We found that while the sanitization is in fact inconsistent, injecting strings in these contexts did not lead to privilege escalation attacks. In part this is because our testing methodology (positive tainting [97]) is conservative: if we cannot prove a string is safe, we flag the path. In other cases, adversary's authority and the policy of the test application made it impossible to exploit the inconsistency.

5.4 Conclusion

We conclude that context-consistent sanitization is error-prone when done manually by developers. Expressive languages, such as those of the .NET platform, permit the use of string operations to construct HTML output as strings with trusted code intermixed with untrusted data. Plus, these rich programming languages allow developers to write complex dataflow and control flow logic. This results in some features of current programming environments, summarised as:

- **String outputs.** String outputs contain trusted constant code fragments mixed with untrusted data.
- **Nested contexts.** Untrusted data is often embedded in nested contexts.

- **Intersecting data-flow paths.** Data variables are used in conflicting or mismatched contexts along two or more intersecting data-flow paths.

- **Custom output controls.** Frameworks such as .NET encourage reusing output rendering code by providing built-in "controls", which are classes that render untrusted inputs in HTML codes. Large applications extensively define custom controls, perhaps because they find the built-in controls insufficient. The running example is typical of such real-world applications — it defines its own custom controls, `DynamicLink`, to render user-specified links via JavaScript.

A large fraction of web application frameworks leave the task of sanitization to developers. Some web frameworks take on the onus of placing sanitizers in application code; however, many of these do not pay attention to a key property of context-sensitive sanitization. Instead, they apply the same sanitizer to all untrusted data outputs which can provide a false sense of security. A small fraction of recent web frameworks support context-sensitive auto-sanitization; however, their security properties and performance characteristics are not evaluated in prior research.

Web Page	Sanitized Paths	Inconsistently sanitized	
		Total	Highlight
Home	396	14	9
A1 P1	565	28	22
A1 P2	336	16	11
A1 P3	992	26	21
A1 P4	297	44	35
A1 P5	482	22	17
A1 P6	436	23	18
A1 P7	403	19	13
A1 P8	255	22	18
A1 P9	214	16	12
A1 P10	1,623	18	14
A2 P1	315	16	12
A2 P2	736	53	47
A2 P3	261	21	16
A2 P4	197	16	12
A2 P5	182	22	18
A2 P6	237	22	18
A2 P7	632	20	16
A2 P8	450	23	19
A2 P9	802	26	22
A3 P1	589	25	21
A3 P2	2,268	18	14
A3 P3	389	16	12
A3 P4	477	103	15
A3 P5	323	24	20
A3 P6	292	51	45
A3 P7	219	16	12
A3 P8	691	25	21
A3 P9	173	16	12
A4 P1	301	24	20
A4 P2	231	30	25
A4 P3	271	28	22
A4 P4	436	38	32
A4 P5	956	36	24
A4 P6	193	24	18
A4 P7	230	36	32
A4 P8	310	24	20
A4 P9	200	24	18
A4 P10	208	24	20
A4 P11	498	34	29
A4 P12	579	34	29
A4 P13	295	25	20
A4 P14	591	104	91
A5 P1	604	61	55
A5 P2	376	25	21
A5 P3	376	25	21
A5 P4	401	26	21
A5 P5	565	31	26
A5 P6	493	34	29
A5 P7	521	34	29
A5 P8	427	24	20
A5 P9	413	24	20
A5 P10	502	28	23
Total	25,209	1,558	1,207

Figure 5.10: Characterization of the fraction of the paths that were inconsistently sanitized. The right-most column indicates paths highlighted as errors by our analysis.

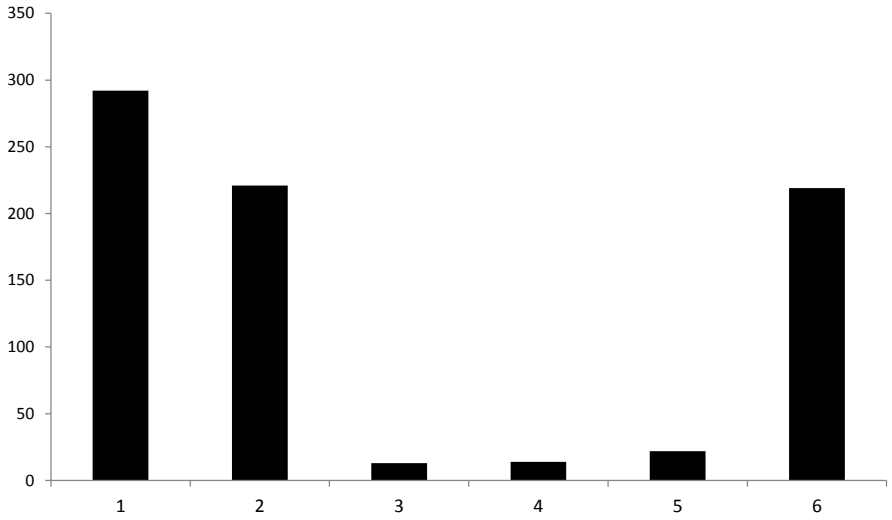

Figure 5.11: Distribution of lengths of paths that could not be proved safe. Each hop in the path is a string propagation function. The longer the chain, the more removed are taint sources from taint sinks.

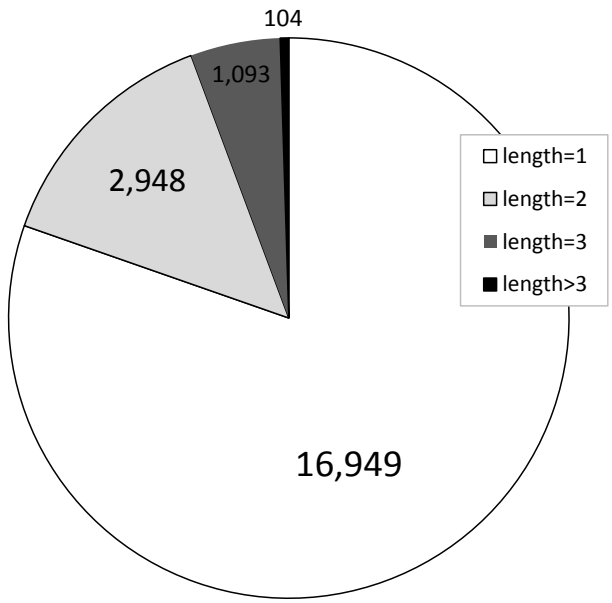

Figure 5.12: Distribution of the lengths of applied sanitization chains, showing a sizable fraction of the paths have more than one sanitizer applied.

Chapter 6

Securing Sanitization-based Defense

In the previous chapter, we outline some of the challenges with employing a sanitization-based defense to scripting attacks. Our observation is that many web application frameworks and real-world applications employ *incorrect placement of sanitizers* in the application code. In this chapter, we investigate a systematic technique to automatically place sanitizers in web application code. We propose a technique that can be utilized by web templating frameworks, though the general idea can be extended to other programming languages too. We begin by introducing web templating frameworks and give an overview of our techniques and its assumptions.

Web Templating Frameworks. To streamline the output generation from application code, numerous web templating frameworks have recently emerged and are gaining widespread adoption [46, 27, 29, 30, 44, 103, 95, 32, 137, 133, 60, 112, 109]. Web templating frameworks allow developers to specify their application's output generation logic in code units or modules called *templates*. Templates take *untrusted inputs* which may be controlled by the attacker and emit web application outputs, such as HTML or CSS code, as strings. String outputs from templates are composed of static or constant strings written by developers, which are explicitly trusted, and untrusted inputs which must be sanitized. These templates can be compiled into a *target language*, such as JavaScript or Java/PHP, as code functions that take untrusted data as template arguments and emit the application's output as strings. Templates are written in a different language, called a *templating language*, the semantics of which are much simpler as compared to that of the target language. Notably, complex constructs such as JavaScript's `eval` and `document.write` are not included in the templating language. Code external to templates is responsible for invoking compiled templates to obtain the string outputs and evaluating/rendering them in the browser.

Vision. Ideally, we would like to create web applications that are secure by construction. In fact, web templating frameworks offer an ideal opportunity to relieve the developers from the burden of manual sanitization by *auto-sanitizing*—inserting sanitization primitives automatically during the compilation of templates to server-side or client-side code. Despite this ideal opportunity, research so far has not broached the topic of building auto-sanitization

defenses in today's commercial templating frameworks.

Challenges. In this work, we first identify the following practical challenges in building reliable and usable auto-sanitization in today's web templating frameworks:

- *Context-sensitivity.* XSS sanitization primitives vary significantly based on the *context* in which the data sanitized is being rendered. For instance, applying the default HTML escaping sanitizer is recommended for untrusted values placed inside HTML tag content context [129]; however, for URL attribute context (such as `src` or `href`) this sanitizer is insufficient because the `javascript` URI protocol (possibly masked) can be used to inject malicious code. We say, therefore, that each sanitization primitive *matches* a context in which it provides safety. Many developers fail to consistently apply the sanitizers matching the context, as highlighted in Chapter 5.

- *Complexity of language constructs.* Templating languages today permit a variety of complex constructs: support for string data-type operations, control flow constructs (`if-else`, loops) and `call`s to splice the output of one template into another. Untrusted input variables may, in such languages, be used in one context along one execution path and a different context along another path. With such rich language features, determining the context for each use of untrusted input variables becomes a path-sensitive, global data-flow analysis task. Automatically applying correct sanitization on all paths in templating code becomes challenging.

- *Backwards compatibility with existing code.* Developers may have already applied sanitizers in existing template code at arbitrary places; an auto-sanitization mechanism should not undo existing sanitization unless it is unsafe. For practical adoption, auto-sanitization techniques should only supplement missing sanitizers or fix incorrectly applied ones, without placing unnecessary restrictions on where to sanitize data.

- *Performance Overhead.* Auto-sanitized templates should have a minimal performance overhead. Previous techniques propose parsing template outputs with a high-fidelity HTML parser at runtime to determine the context [16]. However, the overhead of this mechanism may be high and undesirable for many practical applications.

Context-sensitive Auto-sanitization Problem. We observe that a set of contexts in which applications commonly embed untrusted data is known [125]. And, we assume that for each such context, a matching sanitizer is externally provided. Extensive recent effort has focused on developing a library of safe or correctly-implemented sanitization primitives [66, 99, 129, 69, 53, 51]. We propose to develop an automatic system that, given a template and a library of sanitizers, automatically sanitizes each untrusted input with a sanitizer that matches the context in which it is rendered. By auto-sanitizing templates in this context-sensitive way, in addition to enforcing the security properties we outline in Section 6.1, templating systems can ensure that scripting attacks never result from using template outputs in intended contexts.

Our Approach & Contributions.

In this chapter, we address the outlined challenges with a principled approach:

- *Type Qualifier Based Approach.* We propose a type-based approach to automatically ensure context-sensitive sanitization in templates. We introduce *context type qualifiers*, a kind of type qualifier that represents the *context* in which untrusted data can be safely embedded. Based on these qualifiers, which refine the base type system of the templating language, we define a new type system. Type safety in our type system guarantees that well-typed templates have all untrusted inputs context-sensitively sanitized.

- *Type Inference during Compilation.* To transform existing developer-written templates into well-typed templates, we develop a Context-Sensitive Auto-Sanitiza- tion (CSAS) engine which runs during the compilation stage of a web templating framework. The CSAS engine performs two high-level operations. First, it performs a static type inference to infer context type qualifiers for all variables in templates. Second, based on the inferred context types, the CSAS engine automatically inserts sanitization routines into the generated server-side or client-side code. To the best of our knowledge, our approach is the first principled approach using type qualifiers and type inference for context-sensitive auto-sanitization in templates.

- *Real-world Deployability.* To show that our design is practical, we implement our type system in Google Closure Templates, a commercially used open-source templating framework that is used in large applications such as GMail, Google Plus and Google Docs. Our implementation shows that our approach requires less than 4000 lines of code to be built into an existing commercial web framework. Further, in our experiments we find that retrofitting our type system to existing Google Closure templates used in commercial applications requires no changes or annotations to existing code.

- *Improved Security.* Our approach eliminates the critical drawbacks of existing approaches to auto-sanitization in today's templating frameworks. Though all the major web frameworks today support customizable sanitization primitives, a majority of them today do not automatically apply them in templates, leaving this error-prone exercise to developers. Most others automatically sanitize all untrusted variables with the same sanitizer in a *context-insensitive* manner, a fundamentally unsafe design that provides a false sense of security [125]. Google AutoEscape, the only context-sensitive abstraction we are aware of, does not handle the richness of language features we address. We refer readers to Section 6.7 for a detailed comparison.

- *Fast, Precise and Mostly Static Approach.* We evaluate our type inference system on 1035 existing real-world Closure templates. Our approach offers practical performance overhead of $3 - 9.6\%$ on CPU intensive benchmarks. In contrast, the alternative runtime parsing approach incurs 78% - 510% overhead on the same benchmarks. Our approach performs all parsing and context type inference statically and so

achieves significantly better performance. Our approach does not sacrifice any precision in context-determination as compared to the runtime parsing approach— it defers context-sensitive sanitization to runtime for a small fraction of output operations in which pure static typing is too imprecise. Hence, our type system is *mostly static*, yet precise.

Scope. In this chapter, we limit our study to a small class of web applications written in Google Closure templating framework. We believe the property we define here, namely context-sensitive sanitization, is necessary to achieve safety in sanitization-based defenses but may not be sufficient for all scenarios. For example, web application often vary the sanitization based on the role or authority of the user which controls the sanitized input, application-specific trust policies and so on. We consider these to be out-of-scope of present work. Extending ideas presented in this work to handle such real-world features is a promising direction for future work.

6.1 Problem Definition

The task of auto-sanitization is challenging because state-of-the-art templating frameworks don't restrict templates to be straight-line code. In fact, most templating frameworks today permit control-flow constructs and string data operations to allow application logic to conditionally alter the template output at runtime. To illustrate the issues, we describe a simple templating language that captures the essence of the output-generating logic of web applications. We motivate our approach by showing the various challenges that arise in a running example written in our templating language.

A Simple Templating Language

Our simple templating language is expressive enough to model Google Closure Templates and several other frameworks. We use this templating language to formalize and describe our type-based approach in later sections. It is worth noting that the simple templating language we present here is only an illustrative example—our type-based approach is more general and can be applied to other templating languages as well.

The syntax for the language is presented in Figure 6.1. The templating language has two kinds of data types in its base type system: the primitive (`string, bool, int`) types and a special type (denoted as η) for *output buffers*, which are objects to which templates write their outputs. Figure 6.2(A) shows a running example in our templating language. For simplicity, we assume in our language that there is only a single, global output buffer to which all templates append their output, similar to the default model in PHP.

Command Semantics. The primary command in the language is the `print` command which appends the value of its only operand as a string to the output buffer. The running

Base Types	$\alpha ::=$	$\beta \mid \eta \mid \beta_1 \rightarrow \beta_2 \rightarrow \ldots \beta_k \rightarrow \texttt{unit}$
	$\beta ::=$	$\texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid \texttt{unit}$
Commands	$S ::=$	$\textbf{print}\ (e : \beta)$
		$\mid (v : \beta) := (e : \beta)$
		$\mid \textbf{callTemplate}\ f\ (e_1, \ldots, e_k)$
		$\mid c_1\ ;\ S_1$
		$\mid \textbf{if}(e : \mathit{bool})\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}$
		$\mid \textbf{while}(e : \mathit{bool})\ S_1$
		$\mid \textbf{return};$
Expressions	$e ::=$	$(e_1 : \texttt{int}) \oplus (e_2 : \texttt{int})$
		$\mid (e_1 : \texttt{bool}) \odot (e_2 : \texttt{bool})$
		$\mid (e_1 : \texttt{string}) \cdot (e_2 : \texttt{string})$
		$\mid \textbf{const}\ (i : \beta)$
		$\mid v : \beta$
		$\mid \textbf{San}\ (f, e : \beta)$
	$v ::=$	$Identifier$

Figure 6.1: The syntax of a simple templating language. $\oplus$ represents the standard integer and bitvector arithmetic operators, $\odot$ represents the standard boolean operations and $\cdot$ is string concatenation. The *San* expression syntactically refers to applying a sanitizer.

example has several `print` commands. Note that the syntax ensures that the output buffer (η-typed object) can not be reassigned, or tampered with in the rest of the command syntax.

Templates are akin to functions: they can call or invoke other templates via the `callTemplate` command. This command allows a template to invoke another template during its execution, thereby splicing the callee's outputs into its own. Parameter passing follows standard pass-by-value semantics.

The templating language allows control-flow commands such as `for` and `if-then-else` to allow dynamic construction of template outputs. It supports the usual boolean and integer operations as well as string concatenation. We exclude more complex string manipulation operations like string substitution and interpolation functions from the simple language; with simple extensions, their semantics can be modeled as string concatenations [99].

Restricting Command Semantics. The semantics of the templating language is much simpler than that of a general-purpose language that templates may be compiled to. Notably, for instance, the templating language does not have any dynamic evaluation commands such as JavaScript's `eval` or `document.write`. Therefore, final code evaluation in DOM evaluation constructs or serialization to the HTTP response stream is performed by external application code. For instance, Figure 6.3 below shows a JavaScript application code written outside the templating language which invokes the function compiled from the running example template. It renders the returned result string dynamically using a `document.write`. Therefore, the template code analysis does not need to model the complex semantics of

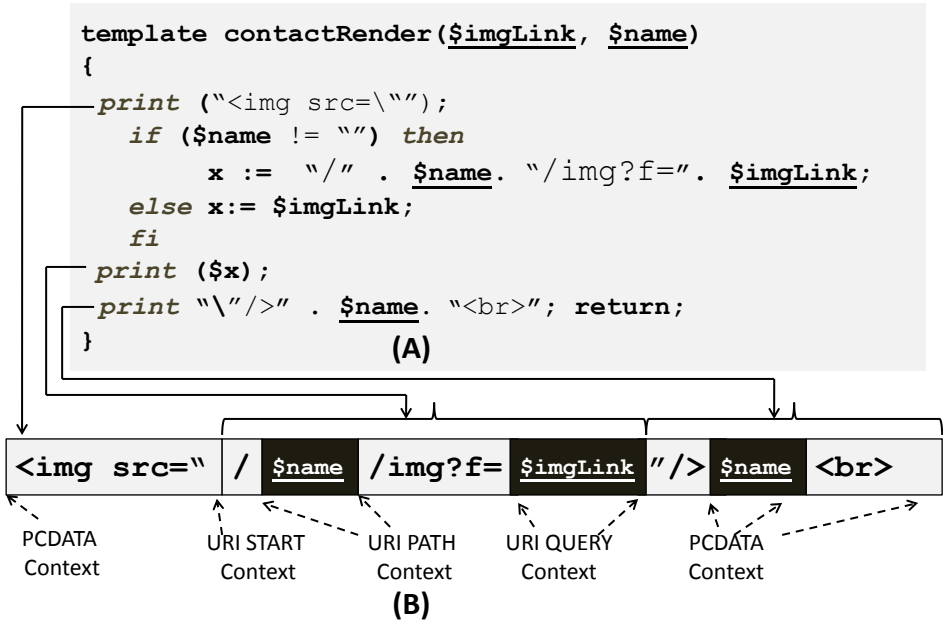

Figure 6.2: (A) shows a template used as running example. (B) shows the output buffer after the running example has executed the path including the true branch of the `if` statement.

`document.write`. [1]

```
<script>
var o = new soy.StringBuilder();
contactRender({O: o, imglink: $_GET('extlink'),
               name: [$_GET('name')] }));
document.write(o);
</script>
```

Figure 6.3: Pseudo-code of how external application code, such as client-side Javascript, can invoke the compiled templates.

Problem Definition & Security Properties

In this chapter, we focus on the following problem: given a templating language such as the one in Section 6.1, and a set of correct sanitization routines for different contexts, the goal is

[1]The semantics of `document.write` varies based on whether the `document` object is open or closed.

to automatically apply the correct sanitization primitives during compilation to all uses of untrusted inputs in constructing template outputs, while satisfying the following properties.

Property `NOS`: No Over-Sanitization. The templating language allows string expressions to be emitted at `print` operations. String expressions may be constructed by concatenation of constant/static strings and untrusted input variables; only the latter should be sanitized or else we risk breaking the intended structure of the template output. For instance in our running example, the auto-sanitization engine should *not* place a sanitizer at the statement `print ($x)`, because the expression `x` consists of a constant string as well as untrusted input value. Sanitizing at this print statement may strip out the `/` or `?` characters rendering the link unusable and breaking the intended structure of the page.

Property `CSAN`: Context-Sensitive Sanitization. Each untrusted input variable should be sanitized with a sanitizer matching the context in which it is rendered in. However, this is challenging because untrusted inputs may be used in two different contexts along two different paths. In our running example, the `$imgLink` variable is used both in a URI context as well as a HTTP parameter context, both of which have different sanitization requirements. Similarly, untrusted inputs can be rendered in two different contexts even along the same path, as seen for the variable `$name` in Figure 6.2 (B). We term such use of inputs in multiple contexts as a *static context ambiguity*, which arise because of path-sensitive nature of the template output construction logic and because of multiple uses of template variables. Section 6.3 describes further scenarios where context ambiguity may arise.

Property `CR`: Context Restriction. Template developers should be forbidden from mistakenly using untrusted values in contexts other than ones for which matching sanitizers are available. Certain contexts are known to be hard to sanitize, such as in an unquoted JavaScript string literal placed directly in a JavaScript eval [94], and thus should be forbidden.

Determining Final Output Start/End Context. For each template, we infer the contexts in which the template's output can be safely rendered. However, since the final output is used external to the template code, providing a guarantee that external code uses the output in an intended context is beyond the scope of our problem. For example, it is unsafe for external code to render the output of the running example in a JavaScript `eval`, but such properties must be externally checked.

Motivation for Our Approach

If a templating language has no control-flow or `callTemplate` constructs and no constructs to create string expressions, all templates would be straight-line code with `print`s of constant strings or untrusted variables. Auto-sanitizing such templates is a straight-forward 3-step process— (a) parse the template statically using a high-fidelity parser (like HTMLPurify [53]), (b) determine the context at each `print` of untrusted inputs and (c) apply the matching sanitizer to it. Unfortunately, real templating languages are often richer like our templating language and more sophisticated techniques are needed.

One possible extension of the approach for straight-line code is to defer the step of parsing and determining contexts to runtime execution [16]. We call this approach a *context-sensitive runtime parsing* (or CSRP) approach, where a parser parses all output from the compiled template, determines the context of each `print` of untrusted input and sanitizes it at runtime. This approach has additional performance overhead due to cost of parsing all application output at runtime, as previously shown [16] and as we evaluate in Section 6.6. If string operations are supported in the language, the performance penalty may be exacerbated because of the need for tracking untrusted values during execution.

Instead, we propose a new "mostly static" approach which off-loads expensive parsing steps to a static type analysis phase. Contexts for most uses of untrusted data can be statically determined and their sanitizers can be selected during compile-time; only a small fraction need the more expensive CSRP-like sanitizer selection in our approach—hence our approach is "mostly static".

Assumptions. Our type-based approach relies on a set of assumptions which we summarize below:

1. *Canonical Parser.* To reliably determine the contexts in which untrusted inputs are rendered, constant/static strings in templates must parse according to a canonical grammar which reliably parses in the same way across major browsers. This restriction is necessary to ensure that our context determination is consistent with its actual parsing in the client's browser, which is challenging because browser parsing behaviors vary in idiosyncratic ways. In our approach, templates not complying with our canonical grammar do not typecheck as per our type rules defined in section 6.3. Google AutoEscape based frameworks such as GWT and CTemplate already tackle the practical issue of developing such a canonical grammar [46, 30, 27]; our engine leverages this existing code base.

2. *Sanitizer Correctness.* As mentioned previously, we assume that a set of contexts in which applications commonly render untrusted inputs is known and their matching sanitizers are externally available. Creating sanitizers that work across major browser versions is an orthogonal challenge being actively researched [51, 53].

3. *End-to-End Security.* As explained earlier, if the external code renders the template outputs in an unintended context or tampers with the template's output before emitting it to the browser, the end-to-end security is not guaranteed. Ensuring correctness of external code that uses template outputs is beyond the scope of the problem we focus here—lint tools, static analysis and code conformance testing can help enforce this discipline externally.

6.2 Our Approach

In our type-based approach, we enforce the aforementioned security properties by attaching or qualifying variables and expressions in templates with a new kind of qualifier which we call the *context type qualifier*. Type qualifiers are a formal mechanism to extend the basic type safety of langauge to enforce additional properties [38]. Context type qualifiers play different roles for the various expressions they qualify. For an untrusted input variable, the context type qualifier captures the contexts in which the variable can be safely rendered. An untrusted input becomes safe for rendering in a certain context only after it is sanitized by a sanitizer matching that context. Unsanitized inputs have the `UNSAFE` qualifier attached, and are not safe to be a part of any expression that is used in a `print` statement. For constant/static string expressions, context type qualifiers capture the result of parsing the expression, that is, the start context in which the expression will validly parse and the context that will result after parsing the expression. When the template code constructs an output string expression by concatenating a constant string and an untrusted input, a type rule over context qualifiers of the two strings ensures that the untrusted input is only rendered in contexts for which it is sanitized.

This rule only enforces the `CSAN` property in the concatenation operation. Several additional rules are needed to enforce all the outlined security properties to cover all operations in our templating language. We describe the full type system with formal type rules over context type qualifiers in section 6.3. The type safety of the type system implies that the security properties outlined in Section 6.1 are enforced.

CSAS Engine. The input to our auto-sanitization engine is an existing template which may be completely devoid of sanitizers. We call these templates *untyped* or vanilla templates. The task of our auto-sanitization engine is two-fold: (a) to convert untyped or vanilla templates into an internal representation (or IR) complying with our type rules (called the well-typed IR), and (b) to compile the well-typed IR to the target language code with sanitization. We develop a CSAS engine in the compiler of a templating framework to handle these tasks. Figure 6.4 shows the CSAS architecture. It has two high-level steps: (A) Type Qualifier Inference, and (B) Compilation of CSAS templates.

The qualifier inference step transforms the vanilla template into a well-typed IR and automatically infers the type qualifiers for all program expressions in the IR. The inferred type qualifiers exactly determine where and which sanitizers are required for untrusted inputs. The well-typed IR must conform to the type rules that we define in Section 6.3. The step (B) compiles the well-typed IR and inserts sanitization primitives and additional instrumentation in the final compiled code. The detailed design of the CSAS engine is presented in Section 6.4.

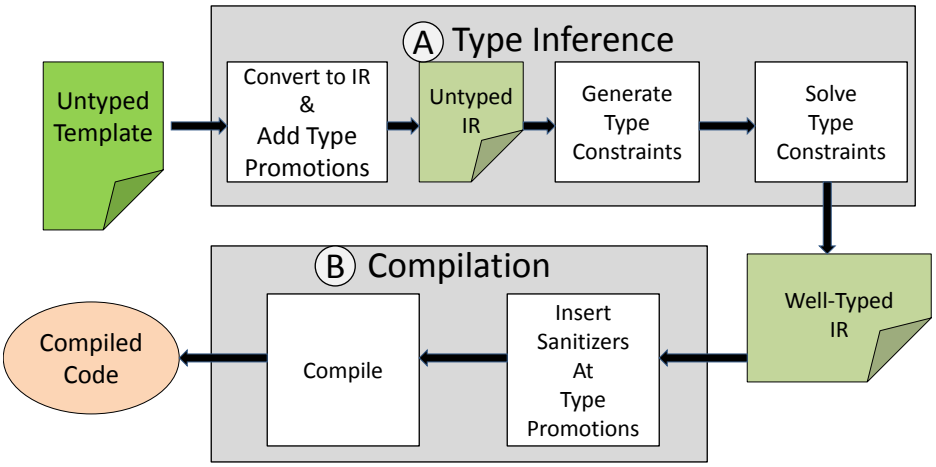

Figure 6.4: Overview of our CSAS engine.

6.3 The Context Type System

In this section, we formally describe our type qualifier mechanism that refines the base type system of the language defined in Section 6.1. We define the associated type rules to which well-typed IR code must conform after the type inference step.

Key Design Points

The CSAS engine must track the parsing context induced by the application's output at each program point. Each string expression, when parsed by the browser, causes the browser to transition from one context to another. We term this transition induced on the browser by parsing strings as a *context transition*, denoted by the notation $c_1 \hookrightarrow c_2$.

Notion of Context Type Qualifiers. Qualifiers play different roles for different kinds of expressions.

For constant/static strings, the context type qualifier captures the context transition it induces when parsed by our canonical grammar. For example, the constant string `<a href="` is qualified with a $\texttt{STATIC}_{\texttt{PCDATA}\hookrightarrow\texttt{URI\_START}}$ context type qualifier indicating that it parses validly as per our canonical grammar (HTML 5), causing the browser to transition from `PCDATA` context (start of tag) to the `URI_START` context. Unsanitized input variables are by default qualified `UNSAFE`. The type system qualifies them with a $\texttt{STATIC}_{c\hookrightarrow c'}$ context qualifier, where c and c' are contexts, only after the variable is sanitized with a sanitizer matching the context c. The sanitizer ensures that the untrusted input safely renders in context c—we define this correctness property of the sanitizer more precisely in Section 6.3. Variables qualified `UNSAFE` are not permitted to be used in string expressions that are emitted

Types	$\tau ::=$	$\mathcal{Q}_1\ \beta \mid \mathcal{Q}_2\ \eta$
Base Types	$\alpha ::=$	$\beta \mid \eta \mid \beta_1 \to \beta_2 \to \ldots \texttt{unit}$
	$\beta ::=$	$\texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid \texttt{unit}$
Type	$\mathcal{Q} ::=$	$\mathcal{Q}_1 \mid \mathcal{Q}_2 \mid \vec{\mathcal{Q}_1} \to [Q_2 \to Q_2]$
Qualifiers	$\mathcal{Q}_1 ::=$	$\texttt{UNSAFE}$
		$\mid \texttt{STATIC}_{c_1 \hookrightarrow c_2} \quad c_1, c_2 \in \mathcal{C}$
		$\mid \texttt{DYN}_{SC} \quad SC \in 2^{\mathcal{C}\times\mathcal{C}}$
	$\mathcal{Q}_2 ::=$	$\texttt{CTXSTAT}_c \quad c \in \mathcal{C}$
		$\mid \texttt{CTXDYN}_S \quad S \in 2^{\mathcal{C}}$
Contexts	$\mathcal{C} ::=$	$\texttt{PCDATA} \mid \texttt{RCDATA} \mid \ldots$

Figure 6.5: The final types τ are obtained by augmenting base types of the language α with type qualifiers $\mathcal{Q}$

to the output buffer.

When data is emitted to the output buffer, the analysis engine must track which context the output buffer is in. The global output buffer (base type η) is also qualified with a different set of context type qualifiers, which indicate the context it is in at any given point in the program. For instance, the output buffer which is in a `URI_START` context, say just after `<a href="` is written to it, is annotated with a $\texttt{CTXSTAT}_{\texttt{URI\_START}}$ context qualifier. The context-sensitivity property is enforced by matching the context type qualifiers of the output buffer and the string expression being written at each `print` command. For example, when an untrusted variable is emitted to a $\texttt{CTXSTAT}_c$ qualified buffer, it must have the $\texttt{STATIC}_{c \hookrightarrow c'}$ qualifier attached, ensuring that it has been sanitized (or made safe) for rendering in context c.

We formally define the qualifiers in Figure 6.5. As explained, the type system defines two separate sets of type qualifiers: $\mathcal{Q}_1$ and $\mathcal{Q}_2$. Type qualifiers $\mathcal{Q}_2$ annotate the output buffer, which is an object of base type η, whereas the set $\mathcal{Q}_1$ exclusively qualifies other typed expressions. Type qualifiers of the form $\vec{\mathcal{Q}_1} \to [Q_2 \to Q_2]$ are inferred for each template function, which capture the expected qualifier types for the arguments and the template's effect on the output buffer, as explained in further detail in Section 6.3.

```
template StatAmb($imgLink, $name)
{
   if ($name == "") then print ("<img src=\"");
   else print ("<div>"); fi
   print ($imgLink);
   if ($name == "") then print ("\" />");
   else print ("</div>"); fi; return;
}
```

Figure 6.6: An example template requiring a mixed static-dynamic approach.

Handling Context ambiguity with Flow-Sensitivity. Untrusted inputs may be used in different contexts along different or even the same program paths. This leads to context ambiguity, as explained in Section 6.1. A standard flow-insensitive type inference algorithm would infer that such an untrusted input has no single precise context qualifier because of its ambiguous usage in multiple different contexts. To handle such context ambiguity, we design our type system to be *flow-sensitive*— a flow-sensitive type system permits program variables to have varying type qualifiers at different program locations [38].

Mixed Static-Dynamic Typing. Flow-sensitive typing does address static ambiguity to a large extent, but not all of the cases we observe in practice. Consider a template such as the one in Figure 6.6. In one branch the program writes `<div>` and in the other it writes `<img src="` to the global output buffer. The context that output buffer is in at the join point is statically ambiguous, and consequently, statically selecting sanitizers on subsequent `print` statements in the template is not possible. Similar examples of static ambiguity have been shown to arise in large legacy applications [97].

Our approach avoids throwing type errors for such static ambiguous types by using the following approach: we further divide the type qualifiers into *statically-qualified* and *dynamically-qualified* sets. Qualifiers $\mathcal{Q}_2$ for the output buffer are either static qualifiers ($\texttt{CTXSTAT}_C$) or dynamic ($\texttt{CTXDYN}_S$). At a given program location, if the output buffer is unambiguously determined to be in a single context c, a static qualifier is attached to it. In contrast, when the embedding context of the buffer is statically ambiguous (or imprecise), as in the example of Figure 6.6, it is over-approximated by a set of contexts S and is qualified with the dynamic qualifer $\texttt{CTXDYN}_S$. $\texttt{CTXDYN}_S$ signifies that the buffer is in one of the contexts determined by the set S. Sanitizers can be statically selected for statically-qualified objects since their contexts are precisely known. For dynamically-qualified buffers, the context-sensitive runtime parsing (or CSRP) approach is employed—data written to such buffers is parsed and sanitized at runtime.

Qualifiers $\mathcal{Q}_1$ for other program expressions are similarly partitioned into static or dynamic sets—for instance, a string expression used in a `print` statement with a dynamically-qualified output buffer is also dynamically-qualified in our type system using the $\texttt{DYN}_S$ qualifier. The set S is a static over-approximation of the set of context transitions that the string expression can induce. Sanitizer selection can be done statically for statically-qualified (such

$$\frac{v \in \mathcal{V} \quad \{v \mapsto Q\} \in \Gamma}{\Gamma \vdash v : Q}\text{T-VAR} \qquad \frac{\alpha_i \neq \texttt{string} \quad c \in \mathcal{C}}{\Gamma \vdash \textbf{const}(i : \alpha_i) : \texttt{STATIC}_{c \hookrightarrow c}}\text{T-CONST}$$

$$\frac{IsParseValid(s, c_1, c_2)}{\Gamma \vdash \textbf{const}(s : \texttt{string}) : \texttt{STATIC}_{c_1 \hookrightarrow c_2}}\text{T-CONSTSTR}$$

$$\frac{\Gamma \vdash e_1 : \texttt{STATIC}_{c \hookrightarrow c} \quad \Gamma \vdash e_2 : \texttt{STATIC}_{c \hookrightarrow c} \quad c \in \mathcal{C}}{\Gamma \vdash (e_1 : \texttt{bool}) \odot (e_2 : \texttt{bool}) : \texttt{STATIC}_{c \hookrightarrow c}}\text{T-BOOL}$$

$$\frac{\Gamma \vdash e_1 : \texttt{STATIC}_{c \hookrightarrow c} \quad \Gamma \vdash e_2 : \texttt{STATIC}_{c \hookrightarrow c} \quad c \in \mathcal{C}}{\Gamma \vdash (e_1 : \texttt{int}) \oplus (e_2 : \texttt{int}) : \texttt{STATIC}_{c \hookrightarrow c}}\text{T-INT}$$

$$\frac{\Gamma \vdash e_1 : \texttt{STATIC}_{c_1 \hookrightarrow c_2} \quad \Gamma \vdash e_2 : \texttt{STATIC}_{c_2 \hookrightarrow c_3}}{\Gamma \vdash (e_1 : \texttt{string}) \cdot (e_2 : \texttt{string}) : \texttt{STATIC}_{c_1 \hookrightarrow c_3}}\text{T-STRCAT-STAT} \quad \frac{\Gamma \vdash e : \texttt{UNSAFE} \quad SanMap(c_1 \hookrightarrow c_2, f) \quad c_1, c_2 \in \mathcal{C}}{\Gamma \vdash \textbf{San}(f, e) : \texttt{STATIC}_{c_1 \hookrightarrow c_2}}\text{T-SAN}$$

$$\frac{IsParseValid(s, c_1, c_2)}{\Gamma \vdash \textbf{const}(s : \texttt{string}) : \texttt{DYN}_{\{c_1 \hookrightarrow c_2\}}}\text{T-CSTRDYN} \qquad \frac{\Gamma \vdash e_1 : \texttt{DYN}_{S_1} \quad \Gamma \vdash e_2 : \texttt{DYN}_{S_2}}{\Gamma \vdash (e_1 : \texttt{string}) \cdot (e_2 : \texttt{string}) : \texttt{DYN}_{S_1 \bowtie S_2}}\text{T-STRCAT-DYN}$$

Figure 6.7: Type Rules for Expressions.

as $\texttt{STATIC}_{c_1 \hookrightarrow c_2}$) expressions and these sanitizers can be placed during compilation. For dynamically-qualified expressions, however, since the context of the output buffer is known only at runtime, the sanitizer selection is performed by the CSRP approach. Specifically, the CSAS engine inserts additional instrumentation for dynamically-qualified string expressions to keep the untrusted substrings in the expression separate from constant substrings. At runtime, when such an expression is being used in a `print`, it is parsed at runtime as per the dynamically-determined start context and the necessary sanitization primitives are applied to the untrusted substrings. In our evaluation, less than 1% of the expressions were dynamically-qualified; a large majority of the cases do not incur the cost of runtime parsing, enabling our type system to be "mostly static".

Handling Context Ambiguity for Templates. Static context ambiguity may manifest for template start and end contexts as well. A template may be invoked in multiple starting contexts or may be expected to return in multiple ending contexts. In such cases, our CSAS engine resolves the ambiguity purely statically, by cloning templates. For templates that may start or end in more than one context, the CSAS engine generates multiple versions of the template during compilation, each specializing to handle a specific pair of start and end contexts.

Inferring Placement of Sanitizers. Our engine can insert sanitizers into code in which developers have manually applied some sanitizers (chosen from the sanitization library), without undoing existing sanitization if it is correct. Our type rules require additional sanitizers to only be inserted at `print` statements and at *type promotion* operations. Type promo-

$$\frac{\Gamma \vdash e : Q \qquad v \in \mathcal{V}}{\Gamma \vdash v := e \Longrightarrow \Gamma[v \mapsto Q]}\text{T-ASSIGN} \qquad \frac{\Gamma \vdash e : Q \qquad Q \leq Q'}{\Gamma \vdash v_1 := (Q')e \Longrightarrow \Gamma[v_1 \mapsto Q']}\text{T-PROM} \qquad \frac{\Gamma_0 \vdash c_1 : \Gamma_1 \qquad \Gamma_1 \vdash S : \Gamma_2}{\Gamma_0 \vdash c_1; S \Longrightarrow \Gamma_2}\text{T-SEQ}$$

$$\frac{\Gamma \vdash e : \texttt{STATIC}_{c_1 \hookrightarrow c_2} \qquad \Gamma \vdash \rho : \texttt{CTXSTAT}_{c_1}}{\Gamma \vdash \textbf{print}(e) \Longrightarrow \Gamma[\rho \mapsto \texttt{CTXSTAT}_{c_2}]}\text{T-PRINT-STATIC-1}$$

$$\frac{\Gamma \vdash e : \texttt{DYN}_{S_1} \qquad \Gamma \vdash \rho : \texttt{CTXDYN}_{S_2} \qquad |CDom(S_1, \mathcal{C}) \cap S_2| \neq 0}{\Gamma \vdash \textbf{print}(e) \Longrightarrow \Gamma[\rho \mapsto \texttt{CTXDYN}_{CRange(S_1,S_2)}]}\text{T-PRINT-DYN-2}$$

$$\frac{Q_{\rho'} = \texttt{CTXSTAT}_{c_{\rho'}} \qquad \Gamma \vdash f : (Q_1, Q_2 \ldots Q_k) \to [Q_\rho \to Q_{\rho'}] \qquad c_\rho, c_{\rho'} \in \mathcal{C} \qquad \bigwedge_{i \in \{1 \ldots k\}} (\Gamma \vdash e_i : Q_i) \qquad \Gamma \vdash \rho : Q_\rho \qquad Q_\rho = \texttt{CTXSTAT}_{c_\rho} \qquad \bigwedge_{i \in \{1 \ldots k\}} ((Q_i \leq \texttt{STATIC}_{c_i \hookrightarrow c_{i'}}) \wedge (c_i \in \mathcal{C}) \wedge (c_{i'} \in \mathcal{C}))}{\Gamma \vdash \textbf{callTemplate} f(e_1, e_2, \ldots, e_k) \Longrightarrow \Gamma[\rho \mapsto \texttt{CTXSTAT}_{c_{\rho'}}]}\text{T-CALL}$$

$$\frac{\Gamma \vdash \rho : \texttt{CTXSTAT}_c \qquad c \in \mathcal{C} \qquad \{\ell \mapsto f\} \in \mathcal{LF} \qquad \Gamma \vdash f : (Q_1, Q_2 \ldots Q_k) \to [Q_\rho \to Q_{\rho'}] \qquad Q_{\rho'} = \texttt{CTXSTAT}_c}{\Gamma \vdash \ell : \textbf{return;} \Longrightarrow \Gamma}\text{T-RET-STAT} \qquad \frac{\Gamma \vdash \rho : Q \qquad c \in \mathcal{C} \qquad Q = \texttt{CTXDYN}_S \qquad |S| = 1 \qquad c \in S \qquad \{\ell \mapsto f\} \in \mathcal{LF} \qquad \Gamma \vdash f : (Q_1, Q_2 \ldots Q_k) \to [Q_\rho \to Q_{\rho'}] \qquad Q_{\rho'} = \texttt{CTXSTAT}_c}{\Gamma \vdash \ell : \textbf{return;} \Longrightarrow \Gamma[\rho \mapsto \texttt{CTXSTAT}_c]}\text{T-RET-DYN}$$

$$\frac{\Gamma_0 \vdash S_1 : \Gamma \qquad \Gamma_0 \vdash S_2 : \Gamma}{\Gamma_0 \vdash \textbf{if}(e)\textbf{then}S_1\textbf{else}S_2 \Longrightarrow \Gamma}\text{T-IFELSE} \qquad \frac{\Gamma \vdash S \Longrightarrow \Gamma}{\Gamma \vdash \textbf{while}(e)S \Longrightarrow \Gamma}\text{T-WHILE}$$

Figure 6.8: Type Rules for Commands. The output buffer (of base type η) is denoted by the symbol ρ.

tion operations identify points where expressions need to be converted from `UNSAFE`-qualified types to statically- or dynamically-qualified types. These type promotion commands have the form $v := (Q)e$, where Q is a qualified type which are introduced by the CSAS engine when converting templates into the IR. Note that this design separates the type inference task from the type safety rules — type promotions may be added anywhere in the IR by the type qualifier inference algorithm, as long as the resulting IR conforms to the type rules after inference.

Static Type Rules

In this section, we define a set of type rules which impose static restrictions `S0` - `S4` to acheive the 3 properties (`CR`, `CSAN` and `NOV`) described in Section 6.1.

The type system is subdivided into two main kinds of typing judgements, one for typing language expressions (Figure 6.7) and one for typing language commands (Figure 6.8). In our type rules, Γ denotes the type environment that maps program variables, the output buffer (denoted by the symbol ρ) and template name symbols to qualifiers $\mathcal{Q}$.

In a flow-sensitive type system like ours, type qualifier for variables change from one program location to another. Therefore, typing judgements for the language commands (Figure 6.8) capture the effects of command execution on type environments and have the

form $\Gamma \vdash c \Longrightarrow \Gamma'$. This judgement states that the command c is well-typed under the type environment Γ and its execution changes the type environment Γ to Γ'. The expression typing judgement $\Gamma \vdash e : Q$ is standard: it states that at the given program location, the expression e has a type qualifier Q under the environment Γ. All expressions that are neither statically-qualified nor dynamically-qualified, map to `UNSAFE` in Γ. The set of declared variables $\mathcal{V}$ and a map $\mathcal{LF}$ from statement labels to their enclosing functions are assumed to be pre-computed and available externally.

Defining Sanitizer Correctness. The soundness of our type system relies on the correctness of externally provided sanitizers. To define sanitizer correctness more precisely, we reuse the notion of *valid syntactic forms*, formalized by Su et. al. [107]. A sanitizer f is *correct* for a context transition $c_s \hookrightarrow c_e$, if all strings sanitized with f are guaranteed to parse validly starting in context c_s yielding an end context c_e according to our canonical grammar, and if the sentential forms generated during such a parse are valid syntactic forms as per the application's intended security policy [107]. In other words, sanitized strings can span different contexts, but all the intermediate contexts induceded during parsing untrusted strings should be syntactically confined to non-terminals allowed by the application's policy. We assume that a relation $SanMap$, mapping each possible context-transition to a matching sanitizer, is available externally.

`S0`: No Implicit Type Casts. Our type system separates `UNSAFE`-qualified, statically-qualified and dynamic-qualified types. It does not permit implicit type conversions between them. Type qualifier conversions are *only* permitted through explicit type promotion operations, according to a *promotibility* relation $\leq$ defined in Figure 6.9.

$$\frac{}{q \leq q} \qquad \frac{c \in S \qquad S \in 2^{\mathcal{C}}}{\texttt{CTXSTAT}_c \leq \texttt{CTXDYN}_S} \qquad \frac{c_1, c_2 \in \mathcal{C}}{\texttt{UNSAFE} \leq \texttt{STATIC}_{c_1 \hookrightarrow c_2}} \qquad \frac{S \in 2^{\mathcal{C} \times \mathcal{C}}}{\texttt{UNSAFE} \leq \texttt{DYN}_S}$$

Figure 6.9: The promotibility relation $\leq$ between type qualifiers

Our promotibility relation is different from the standard subtyping relation ($\preceq$)—for example, the following subsumption rule applies in standard subtyping, but our promotibility relation does not adhere to it:

$$\frac{\Gamma \vdash e : Q_s \qquad Q_s \preceq Q_t}{\Gamma \vdash e : Q_t}\text{T-SUB}$$

The static type qualifier-based restrictions `S1` and `S3` defined below together satisfy the no over-sanitization (`NOS`) property. Similarly, `S2` ensures the context restriction (`CR`) property. The `S3` and `S4` together satisfy the context-sensitivity (`CSAN`) property while maintaining strict separation between dynamically-qualified and statically-qualified expressions.

`S1`: **No Sanitization for Constants.** The rules `T-CONST`, `T-CONSTSTR` and `T-CSTRDYN` show that constant string values acquire the type qualifier without any sanitization. These values are program constants, so they are implicitly trusted.

`S2`: **Canonical Parsing.** The qualifier parameters (denoting the context-transitions) for trusted constant strings are inferred by parsing them according to the canonical grammar. We assume the availability of such a canonical grammar (assumption 1 in Section 6.1), embodied in a predicate $IsParseValid$ defined below.

Definition 1 *$IsParseValid$ is a predicate of type $\texttt{string} \times \mathcal{C} \times \mathcal{C} \rightarrow \texttt{bool}$, such that $IsParseValid(s, c_1, c_2)$ evaluates to true if and only if the data string s parses validly as per the assumed canonical grammar starting in context c_1 yielding a final context c_2.*

`S3`: **Safe String Expression Creation.** The rules for concatenation do not permit strings qualified as `UNSAFE` to be used in concatenations, forcing the type inference engine to type promote (and hence sanitize) operands *before* they can be used in concatenation opertions. The `T-STRCAT-STAT` rule ensures that only statically safe strings can be concatenated whereas the `T-STRCAT-DYN` rule constructs dynamically qualified strings. The latter rule conservatively over-approximates the result's dynamic set of context-transitions that could occur at runtime. For over-approximating sets, we define an inner-join $S_1 \bowtie S_2$ as the set of all context transitions $c_1 \hookrightarrow c_2$ such that $c_1 \hookrightarrow c_3 \in S_1$ and $c_3 \hookrightarrow c_2 \in S_2$.

`S4`: **Context-Sensitive Output.** The rules for print commands ensure that the emitted string expression can not be `UNSAFE`-qualified. Further, the type rule `T-PRINT-STATIC-1` ensures that the context type qualifier of the emitted string matches the context of the output buffer, when both of them are statically-qualified.

Only dynamically-qualified strings can be emitted to dynamically qualified output buffers—a strict separation between dynamic and static type qualified expressions is maintained. The `T-PRINT-DYN-2` type rule capture this case. This requires a runtime parsing, as described in section 6.3, to determine the precise context. The static type rules compute the resulting context for the output buffer by an over-approximate set, considering the context-transition sets of two dynamically-qualified input operands. To compute the resulting context set, we define 2 operations over a context-transition set S for a dynamically qualified type $\texttt{DYN}_S$:

$$CDom(S, E) = \{C_i | C_i \hookrightarrow C_e \in S, C_e \in E\}$$

$$CRange(S, B) = \{C_i | C_s \hookrightarrow C_i \in S, C_s \in B\}$$

Control flow Commands. Type rules `T-IFELSE` and `T-WHILE` for control flow operations are standard, ensuring that the type environment Γ resulting at join points is consistent. Whenever static context ambiguity arises at a join point, the types of the incoming values must be promoted to dynamically-qualified type to conform to the type rules. Our type inference step (as Section 6.4 explains) introduces these type promotions at join points in

the untyped IR, so that after type inference completes, the well-typed IR adheres to the `T-IFELSE` and `T-WHILE` rules.

Calls and Returns. In our language, templates do not return values but take in parameters passed by value. In addition, templates have side-effects on the global output buffer. For a template f, Γ maps f by name to a type $(Q_1, Q_2, \ldots Q_k) \rightarrow [Q_\rho \rightarrow Q_{\rho'}]$, where $(Q_1, Q_2, \ldots Q_k)$ denotes the expected types of its arguments and $Q_\rho \rightarrow Q_{\rho'}$ denotes the side-effect of f on the global output buffer ρ. The `T-CALL` rule imposes several restrictions.

First, it enforces that each formal parameter either has a statically-qualified type or is promotible to one (by relation $\leq$). Second, it ensures that the types of actual parameters and the corresponding formal parameters match. Finally, it enforces that each (possibly cloned) template starts and ends in statically precise contexts, by ensuring that Q_ρ and $Q_{\rho'}$ are statically-qualified. The output buffer (ρ) can become dynamically qualified within a template's body, as shown in example of Figure 6.6, but the context of ρ should be precisely known at the return statement. In the example of Figure 6.6, the context of ρ is ambiguous at the join-point of the first if-else block. However, we point out that at the return statement the dynamically qualified set of contexts becomes a singleton, that is, the end context is precisely known. The `T-RET-DYN` rule applies in such cases and soundly converts the qualifier for ρ back to a statically-qualified type.

For templates that do not start and end in precise contexts, our CSAS engine creates multiple clones of the template, as explained in Section 6.4, to force conformance to the type rules.

Sanitization

Handling manually placed sanitizers. The `T-SAN` rule converts the type of the expression e in the sanitization expression $San(f, e)$ from `UNSAFE` to a statically-qualified type $\texttt{STATIC}_{c_1 \hookrightarrow c_2}$, only if f is a correct sanitizer for the context transition $c_1 \hookrightarrow c_2$ according to the externally specified $SanMap$ relation.

Auto-sanitization Only at Type Promotions. Other than `T-SAN`, the `T-PROM` type rule is the only way an `UNSAFE`-qualified string can become statically-qualified. The CSAS engine inserts statically selected sanitizers during compilation only at the type promotion command that promote `UNSAFE`-qualified to statically-qualified strings. For such a command $v := (\texttt{STATIC}_{c_1 \hookrightarrow c_2})e$, the CSAS engine's compilation step automatically inserts the sanitizer which matches the start context c_1 and will ensure that parsing v will safely end in context c_2 .

Type Promotion from `UNSAFE` to Dynamic. For dynamically qualified strings, the CSAS engine needs to perform runtime parsing and sanitization. To enable this for dynamically-qualified strings, our instrumentation uses an auxiliary data structure, which we call the CSRP-expression, which keeps constant substrings separate from the untrusted components. For conceptual simplicity, our CSRP-expression data structure is simply a string in which untrusted substrings are delimited by special characters ⟪⟫. These special delimiters are not

part of the string alphabet of the base templating language.

The `T-PROM` rule permits promotions from `UNSAFE`-qualified strings to dynamically-qualified expressions. The CSAS engine inserts instrumentation during compilation to insert the special characters ⟪⟫ around the untrusted data and to initialize this CSRP-expression with it. The concatenation operation over regular strings naturally extends to CSRP-expressions.

Runtime Parsing and Sanitization. At program points where the output buffer is dynamically-qualified, the CSAS engine adds instrumentation to track its *dynamic context* as a metadata field. The metadata field is updated at each `print`. When a CSRP-expression is written to the output buffer at runtime, the CSRP-expression is parsed starting in the dynamically-tracked context of the output buffer. This parsing procedure internally determines the start and end context of each untrusted substring delimited by ⟪⟫, and selects sanitizers for them context-sensitively.

We detail the operational semantics for the language and sketch the soundness proof for our type system in Secion 6.5.

6.4 CSAS Engine

We present the design and implementation of the CSAS engine in this section. The CSAS engine performs two main steps of inferring context type qualifiers and then compiling well-typed IR to JavaScript or server-side Java code with sanitization logic.

Type Qualifier Inference & Compilation

The goal of the type inference step is to convert untyped or vanilla templates to well-typed IR. In the the qualifier inference step, the CSAS engine first converts template code to an internal SSA representation (untyped IR). The qualifier inference sub-engine is also supposed to add additional type promotions for untrusted inputs, where sanitization primitives will eventually be placed. However, the qualifier inference sub-engine does not *apriori* know where all sanitizations will be needed. To solve this issue, it inserts a set of candidate type promotions, only some of which will be compiled into sanitizers. These candidate type promotions include *type qualifier variables*, i.e., variables whose values are context types and are to be determined by the type inference. They have the form $v' := (Q)e$ where Q is a type qualifier variable, and its exact value is a context type to be determined by the type qualifier inference sub-engine. Next, the type qualifier inference step solves for these qualifier variables by generating type constraints and solving them.

Once constraint solving succeeds, the concrete context type for each qualifier variable is known. These context types can be substituted into the candidate type promotions; the resulting IR is well-typed and is guaranteed to conform to our type rules. In the final compilation step, only some of the candidate type promotions are turned into sanitizer calls. Specifically, type promotions in well-typed IR that essentially cast from a qualified-type to itself, are redundant and don't require any sanitization, whereas those which cast `UNSAFE`-

qualified variables into other qualified values are compiled into sanitizers as described in section 6.3.

Inserting Type Promotions with Qualifer Variables

Candidate type promotions are introduced at the following points while converting templates to the untyped IR:

- Each `print` (e) statement is turned into a `print` (v') statement in the IR by creating a fresh internal program variable v'. The CSAS engine also inserts a type promotion (and assignment) statement $v' := (Q)\ e$ preceeding the print statement, creating a qualifier variable Q.
- Each $v = \phi(v_1, v_2)$ statement is turned into equivalent type promotions $v := (Q_1)\ v_1$ and $v := (Q_2)\ v_2$ in the respective branches before the join point, by creating new qualifier variables Q_1 and Q_2 .
- Parameter marshalling from actual parameter "a" to formal parameter "v" is made explicit via a candidate promotion operation $v := (Q)\ a$, by creating new qualifier variable Q.
- A similar type promotion is inserted before the concatenation of a constant string expression with another string expression.

Constraint Solving for Qualifier Variables

The goal of this step is to infer context type qualifiers for qualifier variables. We analyze each template's IR starting with templates that are used by external code— we call these public templates. We generate a version of compiled code for each start and end context in which a template can be invoked, so we try to analyze each public template for each choice of a start and end context. Given a template T, start context c_s and end context c_e, the generic type inference procedure called $TempAnalyze(T, c_s, c_e)$ is described below.

$TempAnalyze(T, c_s, c_e)$ either succeeds having found a satisfying assignment of qualifier variables to context type qualifiers, or it fails if no such assignment is found. It operates over a call-graph of the templates in depth-first fashion starting with T, memoizing the start and end contexts for each template it analyzes in the process. When analyzing the body of a template in IR form, it associates a typemap $\mathcal{L}$ mapping local variables to type qualifiers at each program location. At the start of the inference for T, all local variables are qualified as `UNSAFE` in $\mathcal{L}$. The analysis proceeds from the entry to the exit of the template body statement by statement, updating the context qualifier of each program variable. The context of the output buffer is also updated with the analysis of each statement.

Type rules defined in Figure 6.8 can be viewed as inference rules as well: for each statement or command in the conclusion of a rule, the premises are type constraints to be satisfied. Similar constraints are implied by type rules for expressions. Our type inference generates

and solves these type constraints during the statement by statement analysis using a custom constraint solving procedure.

Several of our type rules are non-deterministic. As an example, the rules `T-CONSTSTR` and `T-CSTRDYN` have identical premises and are non-deterministic because the language syntax alone is insufficient to separate statically and dynamically qualified types. Our constraint solving procedure resolves such non-determinism by backtracking to find a satisfying solution to the constraints. Our inference prefers the most precise (or static) qualifiers over less precise (dynamic) qualifiers as solutions for all qualifier variables during its backtracking-based constraint solving procedure. For instance, consider the non-determinism inherent in the premise involving $IsParseValid$ used in the `T-CONSTSTR` and `T-CSTRDYN` rules. $IsParseValid$ is a one-to-many relation and a constant string may parse validly in many start contexts. Our constraint solving procedure non-deterministically picks one such possible context transition initially, trying to satisfy all instances of the `T-CONSTSTR` rule before that of the `T-CSTRDYN` rule and refines its choice until it finds a context transition under which the static string parses validly. If no instance of the `T-CONSTSTR` rule matches, the engine tries to satisfy the `T-CSTRDYN` rule. Similar, backtracking is also needed when analyzing starting and ending contexts of templates when called via the `callTemplate` operation.

Resolving Context Ambiguity by Cloning

The static typing `T-CALL` rule for `callTemplate` has stringent pre-conditions: it permits a unique start and end context for each template. A templates can be invoked in multiple different start (or end) contexts—our inference handles such cases while keeping the consistency with the type rules by cloning templates. We memoize start and end contexts inferred for each template during the inference analysis. If during constraint generation and solving, we find that a template T is being invoked in start and end contexts different from the ones inferred for T previously during the inference, we create a clone T'. The cloned template has the same body but expects to begin and end in a different start and end context. Cloned templates are also compiled to separate functions and the calls are directed to the appropriate functions based on the start and end contexts.

6.5 Operational Semantics

We have discussed the static type rules in Section 6.3. In this section, we describe the various runtime parsing checks that our CSAS engine inserts at various operations to achieve type safety. We do this by first presenting a big-step operational semantics for an abstract machine that evaluates our simple templating language. We sketch the proof for the soundness of our type system based on the operational semantics.

The evaluation rules are shown in Fig 6.5. Commands operate on a memory M mapping program variables to *values*. Each premise in an evaluation rule has the form $M \vdash e \Downarrow v$ which means that the expression e evaluates to a final value v under the state of the memory

M. Command evaluation judgements have the form $M \vdash c \Downarrow M'$ which states that under the memory M evaluation of the command c results in a memory M'. We omit the rules for template calls and returns here which follow standard call-by-value semantics for brevity.

Values. The values produced during the evaluation of the language are mainly of two kinds: (a) V_β values for the data elements of base type β and (b) V_η for objects of base type η. Runtime errors are captured by a third, explicit `CFail` value. The syntax of values is described in Figure 6.10.

$$
\begin{array}{ll}
Value ::= & V_\beta | V_\eta | \texttt{CFail} \\
 & \\
V_\beta ::= & \langle Val, CTran \rangle | \triangleleft Expr \triangleright | Val \\
Ctran ::= & \mathcal{C} \hookrightarrow \mathcal{C} \\
Val ::= & b | i | s \\
Expr ::= & Val | Expr \cdot Expr | (\!| Val |\!) \\
 & \\
V_\eta ::= & \| s, EmbCtx \| \\
EmbCtx ::= & \mathcal{C}
\end{array}
$$

Figure 6.10: Syntax of Values

The universe of `string`, `int` or `bool` base typed values is denoted by the letters s,i, and b letters respectively in the syntax above and the standard concatenation operation is denoted by "$\cdot$". The V_β values are of three kinds:

1. Untrusted or unsanitized values are raw untrusted string, integer or boolean values.
2. Other values which are auto-sanitized statically or correspond to program constants are tuples of the form $\langle v, Ctran \rangle$ where $Ctran$ is a metadata field. The $Ctran$ metadata field indicates that the value v safely induces a context transition $Ctran$.
3. The remaining values are a special data-structure $\triangleleft Expr \triangleright$, called the CSRP-expression, which is used for dynamically sanitized values. The data structure stores concatenation expressions, conceptually separating the untrusted peices from trusted peices. In our syntax, we separate untrusted substrings of string expressions by delimiting them with special delimiters, $(\!|$ and $|\!)$, which are assumed to be outside the string alphabet of the base language.

The global output buffer has a value of the form $\| s, EmbCtx \|$, where s is the string buffer consisting of the application's output. The $EmbCtx$ metadata field is the context as a result of parsing s according to our canonical grammar.

Type Safety. Note that the operational semantics ensure the 3 security properties outlined in section 6.1. The `CR` is explicit in the representation of output buffer values—parsing the

$$\frac{c \in \mathcal{C}}{M \vdash \mathbf{const}(n) \Downarrow \langle n, c \hookrightarrow c\rangle}\text{E-CINT} \qquad \frac{c \in \mathcal{C}}{M \vdash \mathbf{const}(b) \Downarrow \langle b, c \hookrightarrow c\rangle}\text{E-CBOOL}$$

$$\frac{IsParseValid(s, c_1, c_2)}{M \vdash \mathbf{const}(s) \Downarrow \langle s, c_1 \hookrightarrow c_2\rangle}\text{E-CONST} \qquad \frac{}{M \vdash \mathbf{const}(s) \Downarrow \langle\!\langle s \rangle\!\rangle}\text{E-STRING-DYN}$$

$$\frac{M \vdash e_1 \Downarrow \langle b_1, c \hookrightarrow c\rangle \quad M \vdash e_2 \Downarrow \langle b_2, c \hookrightarrow c\rangle}{M \vdash e_1 \odot e_2 \Downarrow \langle b_1 \odot b_2, c \hookrightarrow c\rangle}\text{E-BL} \qquad \frac{M \vdash e_1 \Downarrow \langle n_1, c \hookrightarrow c\rangle \quad M \vdash e_2 \Downarrow \langle n_2, c \hookrightarrow c\rangle}{M \vdash e_1 \oplus e_2 \Downarrow \langle n_1 \oplus n_2, c \hookrightarrow c\rangle}\text{E-INT}$$

$$\frac{M \vdash e_1 \Downarrow \langle s_1, c_1 \hookrightarrow c_2\rangle \quad M \vdash e_2 \Downarrow \langle s_2, c_3 \hookrightarrow c_4\rangle \quad c_2 = c_3}{M \vdash e_1 \cdot e_2 \Downarrow \langle s_1 \cdot s_2, c_1 \hookrightarrow c_4\rangle}\text{E-CAT-STAT} \qquad \frac{M \vdash e_1 \Downarrow \langle\!\langle s_1 \rangle\!\rangle \quad M \vdash e_2 \Downarrow \langle\!\langle s_2 \rangle\!\rangle}{M \vdash e_1 \cdot e_2 \Downarrow \langle\!\langle s_1 \cdot s_2 \rangle\!\rangle}\text{E-CAT-DYN}$$

$$\frac{M \vdash e \Downarrow s \quad SanMap(c_1 \hookrightarrow c_2, f) \quad s' = f(s) \quad c_1, c_2 \in \mathcal{C}}{M \vdash \mathbf{San}(f, e) \Downarrow \langle s', c_1 \hookrightarrow c_2\rangle}\text{E-SAN}$$

$$\frac{M \vdash e \Downarrow s \quad SanMap(c_1 \hookrightarrow c_2, f) \quad \underline{s' = f(s)} \quad c_1, c_2 \in \mathcal{C}}{M \vdash v := (\texttt{STATIC}_{c_1 \hookrightarrow c_2})e \Downarrow M[v \mapsto \langle s', c_1 \hookrightarrow c_2\rangle]}\text{E-PROM-ST}$$

$$\frac{M \vdash e \Downarrow s}{M \vdash v := (\texttt{DYN}_S)e \Downarrow M[v \mapsto \langle\!\langle\!\langle s \rangle\!\rangle\!\rangle]}\text{E-PROM-DYN}$$

$$\frac{M \vdash e \Downarrow \langle s, c_1 \hookrightarrow c_2\rangle}{M \vdash v := (\texttt{STATIC}_{c_1 \hookrightarrow c_2})e \Downarrow M}\text{E-CAST-1} \qquad \frac{M \vdash e \Downarrow \langle\!\langle s \rangle\!\rangle}{M \vdash v = (\texttt{DYN}_S)e \Downarrow M}\text{E-CAST-2}$$

$$\frac{M \vdash e \Downarrow \langle s, c_1 \hookrightarrow c_2\rangle \quad M \vdash \rho \Downarrow \|s', c\| \quad c = c_1}{M \vdash print(e) \Downarrow M[\rho \mapsto \|s' \cdot s, c_2\|]}\text{E-PRN-STAT} \qquad \frac{M \vdash e \Downarrow \langle\!\langle s \rangle\!\rangle \quad M \vdash \rho \Downarrow \|s', c\| \quad \boxed{(s'', c_2) = CSRP(c, \langle\!\langle s \rangle\!\rangle)}}{M \vdash print(e) \Downarrow M[\rho \mapsto \|s' \cdot s'', c_2\|]}\text{E-PRN-DYN}$$

$$\frac{M \vdash e \Downarrow true \quad M \vdash S_1 \Downarrow M'}{M \vdash \mathbf{if}(e)\mathbf{then}S_1\mathbf{else}S_2 \Downarrow M'}\text{E-IF} \qquad \frac{M \vdash e \Downarrow false \quad M \vdash S_2 \Downarrow M'}{M \vdash \mathbf{if}(e)\mathbf{then}S_1\mathbf{else}S_2 \Downarrow M'}\text{E-EL}$$

$$\frac{M \vdash e \Downarrow true \quad M \vdash S; \mathbf{while}(e)S \Downarrow M'}{M \vdash \mathbf{while}(e)S \Downarrow M'}\text{E-WHLTRUE} \qquad \frac{M \vdash e \Downarrow false}{M \vdash \mathbf{while}(e)S \Downarrow M}\text{E-WHLFALSE}$$

$$\frac{M \vdash e \Downarrow x}{M \vdash v := e \Downarrow M[v \mapsto x]}\text{E-ASSIGN} \qquad \frac{M \vdash c \Downarrow M' \quad M' \vdash S \Downarrow M''}{M \vdash c; S \Downarrow M''}\text{E-SEQ}$$

Figure 6.11: Operational Semantics for an abstract machine that evaluates our simple templating language.

output buffer at any point results in a permitted context defined in $\mathcal{C}$. Property `NOS` is similarly ensured in the `E-CONST` rule, which evaluate to values of the form $\langle v, Ctran\rangle$. Such

values are never sanitized.

Property `CSAN` is immidiate for `E-PRN-STAT` evaluation rule which ensures that the output buffer's context matches the start context in the $Ctran$ field of the written string expression. For the evaluation rule `E-PRN-DYN`, the soundness relies on the procedure $CSRP$ shown boxed which takes a CSRP-expression and a start context to parse the expression in. This procedure parses the string expression embedded in the CSRP-expression, while sanitizing all and only the untrusted substrings delimited by ⟪⟫ context-sensitively. If it succeeds it returns a tuple containing the sanitized string expression and the end context.

In order to formalize and sketch the proof of the soundness of our type system, we first define a relation $\mathcal{R}$ mapping types to the set of valid values they correspond to. At any given point in the program, if the type of a variable or object is Q under the typing environment Γ, then its value must correspond to the typing constraints. The relation $\mathcal{R}$ is defined as follows, assuming $\mathcal{U}$ is the universe of strings, integer and boolean values:

Definition 2 *(**Relation** $\mathcal{R}$)*

$$\begin{array}{ll}
\mathcal{R}(\texttt{UNSAFE}) & = \{v | v \in \mathcal{U}\} \\
\mathcal{R}(\texttt{STATIC}_{c1 \hookrightarrow c2}) & = \{\langle v, c1 \hookrightarrow c2\rangle | v \in \mathcal{U}, IsParseValid(v, c_1, c_2)\} \\
\mathcal{R}(\texttt{DYN}_S) & = \{\triangleleft v \triangleright | v \in \mathcal{U}\} \\
\mathcal{R}(\texttt{CTXSTAT}_c) & = \{\|s, c\| | c \in \mathcal{C}\} \\
\mathcal{R}(\texttt{CTXDYN}_S) & = \{\|s, c\| | c \in \mathcal{C}, c \in S\}
\end{array}$$

At any program location, we define a notion of a well-typed memory M as follows:

Definition 3 *(**Well-Typedness**) A memory is well-typed with respect to a typing environment Γ, denoted by $M \models \Gamma$, iff*

$$\forall x \in Dom(M), M[x] \in \mathcal{R}(\Gamma[x])$$

We define the two standard progress and preservation theorems that establish soundness our type system below. Progress states that if the memory if well-typed, then the abstract machine defined in the operational semantics does not get "stuck"— that is, there is at least one evaluation rule that can be applied to the well-typed terms. Preservation states that at any evaluation step if all the subterms (used in the premises) are well-typed then the deduced term is also well-typed.

The machine may get stuck for several reasons. It may be due to the runtime boxed checks failing. This is intended semantics of the language and such behavior is safe. The machine may also get stuck because no evaluation rule may apply, or in other words, the memory state is such that the semantics do not define how to evaluate further. To distinguish these two cases, we define a value called `CFail`, which the boxed procedure $CSRP$ evaluates to when it fails. Other stuck states may result from reaching an inconsistent memory states for which no evaluation rule applies. We point out the abstract operational semantics we describe here are non-deterministic. Therefore, only when all non-deterministic evaluations of an instance of the procedure $CSRP$ fail, does the boxed check fail and the `print` statement evaluate to the `CFail` runtime error.

Theorem 1 *(Progress and Preservation for Expressions). If* $\Gamma \vdash e : T$ *and* $M \models \Gamma$*, then either* $M \vdash e \Downarrow \texttt{CFail}$ *or* $M \vdash e \Downarrow v$ *and* $v \in \mathcal{R}(T)$*.*

Proof: By induction on the derivation of $\Gamma \vdash e : T$. ■

Theorem 2 *(Progress and Preservation for Commands). If* $M \models \Gamma$ *and* $\Gamma \vdash c \Longrightarrow \Gamma'$*, then either* $M \vdash c \Downarrow \texttt{CFail}$ *or* $M \vdash c \Downarrow M'$ *where* $M' \models \Gamma'$*.*

Proof: By induction on the derivation of $\Gamma \vdash c \Longrightarrow \Gamma'$. The definition of relation $\mathcal{R}$ serves as the standard inversion lemma. ■

6.6 Implementation & Evaluation

We have implemented our CSAS engine design into a state-of-the-art, commercially used open-source templating framework called Google Closure Templates [44]. Closure Templates are used extensively in large web applications including Gmail, Google Docs and other Google properties. Our auto-sanitized Closure Templates can be compiled both into JavaScript as well as server-side Java code, enabling building reusable output generation elements.

Contexts
HTML PCDATA
HTML RCDATA
HTML TAGNAME
HTML ATTRIBNAME
QUOTED HTMLATTRIB
UNQUOTED HTMLATTRIB
JS STRING
JS REGEX
CSS ID, CLASS, PROPNAME, KEYWDVAL, QUANT
CSS STRING, CSS QUOTED URL, CSS UNQUOTED URL
URL START, URL QUERY, URL GENERAL

Figure 6.12: A set of contexts $\mathcal{C}$ used throughout the chapter.

Our implementation is in 3045 lines of Java code, excluding comments and blank lines, and it augments the existing compiler in the Closure Templates with our CSAS engine. All the contexts defined in Figure 6.12 are supported in the implementation with 20 distinct sanitizers.

Subject Benchmarks. For real-world evaluation, we gathered all Closure templates accessible to us. Our benchmarks consist of 1035 distinct Closure templates from Google's commercially deployed applications. The templates were authored by developers prior to

our CSAS engine implementation. Therefore, we believe that these examples represent unbiased samples of existing code written in templating languages.

The total amount of code in the templates (excluding file prologues and comments outside the templates) is $21,098$ LOC. Our benchmarks make heavy use of control flow constructs such as `callTemplate` calls. Our benchmark's template call-graph is densely connected. It consists of 1035 nodes, 2997 call edges and 32 connected components of size ranging from 2 - 12 templates, one large component with 633 templates and 262 disconnected templates (that make no external calls or get called). Overall, these templates have a total of 1224 print statements which write untrusted data expressions. The total number of untrusted input variables in the code base is 600, ranging from $0 - 13$ for different templates.

Evaluation Goals. The goal of our evaluation is to measure how easily our principled type-based approach retrofits to an existing code base. In addition, we compare the security and performance of our "mostly static", context-sensitive approach to the following alternative approaches:

- *No Auto-Sanitization.* This is the predominant strategy in today's web frameworks.
- *Context-insensitive sanitization.* Most remaining web frameworks supplement each output `print` command with the same sanitizer.
- *Context-sensitive runtime parsing sanitization.* As explained earlier, previous systems have proposed determining the contexts by runtime parsing [16]. We compare the performance of our approach against this approach.

Compatibility & Precision

Our benchmark code was developed prior to our type system. We aim to evaluate the extent to which our approach can retrofit security to existing code templates. To perform this experiment, we disabled all sanitization checks in the benchmarks that may have been previously applied and enabled our auto-sanitization on all of the 1035 templates. We counted the fraction of templates that were transformed to well-typed compiled code. Our analysis is implemented in Java and takes 1.3 seconds for all the 1035 benchmarks on a platform with 2 GB of RAM and an Intel 2.6 MHz dual-core processor running Linux 2.6.31.

Our static type inference approach avoids imprecision by cloning templates that are called in more than one context. In our analysis, 11 templates required cloning which resulted in increasing the output `print` statements (or sinks) from 1224 initially to 1348 after cloning.

Our main result is that all 1348 output sinks in the 1035 templates were auto-sanitized. No change or annotations to the vanilla templates were required. We test the outputs of the compiled templates by running them under multiple inputs. The output of the templates under our testing was unimpacted and remained completely compatible with that of the vanilla template code.

Our vanilla templates, being commercially deployed, have existing sanitizers manually applied by developers and are well-audited for security by Google. To confirm our compat-

	No Sanitization	Context-Insensitive	Context-Sensitive Runtime Parsing	Our Approach
Chrome 9.0	227	234 (3.0%)	406 (78.8%)	234 (3.0%)
FF 3.6	395	433 (9.6%)	2074 (425%)	433 (9.6%)
Safari 5.0	190	195 (2.5%)	550 (189%)	196 (3.1%)
Server:Java	431	431 (0.0%)	2972 (510%)	431 (0.0%)
# of Sinks Auto-Prot.	0/ 1348 (0%)	982 / 1348 (72%)	1348 / 1348 (100%)	1348 / 1348 (100%)

Figure 6.13: Comparing the runtime overhead for parsing and rendering the output of all the compiled templates in milliseconds. This data provides comparison between our approach and alternative existing approaches for server-side Java and client-side JavaScript code generated from our benchmarks. The percentage in parenthesis are calculated over the base overhead of no sanitization reported in the second column. The last line shows the number of sinks auto-protected by each approach—a measure of security offered by our approach compared to its alternatives.

ibility and correctness, we compared the sanitizers applied by our CSAS engine to those pre-applied in the vanilla versions of the benchmarked code manually by developers. Out of the 1348 print statements emitting untrusted expressions, the sanitization primitives on untrusted inputs exactly match the pre-applied sanitizers in all but 21 cases. In these 21 cases, our CSAS engine applies a more accurate (`escapeHtmlAttribute`) sanitizer versus the more restrictive sanitizer applied previously (`escapeHTML`) by the developer. Both sanitizers defeat scripting attacks; the pre-existing sanitizer was rendering certain characters inert that weren't dangerous for the context. This evaluation strengthens our confidence that our approach does not impact/alter the compatibility of the HTML output, and that our CSAS engine implementation applies sanitization correctly.

Our type qualifier inference on this benchmark statically-qualified expressions written to all but 9 out of the 1348 sinks. That is, for over 99% of the output sinks, our approach can statically determine a single, precise context.

In these 9 cases, the set of ambiguous contexts is small and a single sanitizer that sanitizes the untrusted input for all contexts in this set can be applied. In our present implementation, we have special-cased for such cases by applying a static sanitizer, which is safe but may be over-restrictive. We have recently implemented the CSRP scheme using an auxiliary data structure, as described in Section 6.3, in jQuery templates for JavaScript [92]; we expect porting this implementation to the Google Closure compiler to be a straight-forward task in the future.

Security

To measure the security offered by our approach as compared to the context-insensitive sanitization approach, we count the number of sinks that would be auto-sanitized correctly

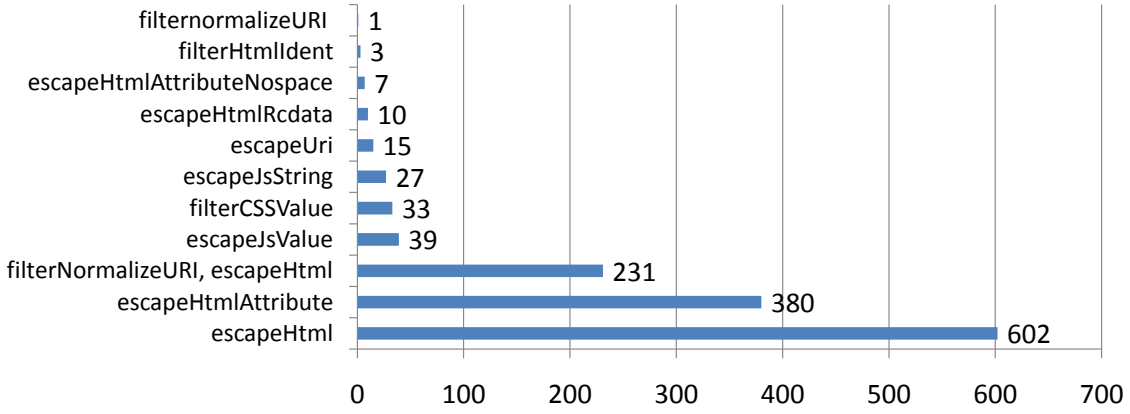

Figure 6.14: Distribution of inserted sanitizers: inferred contexts and hence the inserted sanitizer counts vary largely, therefore showing that context-insenstive sanitization is insufficient.

in our 1035 templates. We assume that a context-insensitive sanitization would apply the HTML-entity encoding sanitizer to all sinks, which is the approach adopted in popular frameworks such as Django [32]. Picking another sanitizer would only give worse results for the context-insensitive scheme— we show that the most widely inserted sanitizer in auto-sanitization on our benchmarks is `escapeHtml`, the HTML-entity encoding sanitizer.

The last row in Figure 6.13 shows the number of sinks auto-protected by existing approaches. Context-insensitive sanitization protects 72% of the total output prints adequately; the auto-sanitization is insufficient for the remaining 28% output `print` opertions. Clearly, context-insensitive sanitization offers better protection than no sanitization strategy. On the other hand, context-sensitive sanitization has full protection whether the context-inference is performed dynamically or as in our static type inference approach. Figure 6.14 shows that the inferred sanitizers varied significantly based on context across the 1348 output points, showing the inadequacy of context-insensitive sanitization.

Performance

We measure and compare the runtime overhead incurred by our context-sensitive auto-sanitization to other approaches and present the results in Figure 6.13. Google Closure Templates can be compiled both to JavaScript as well as Java. We measure the runtime overhead for both cases and report the rendering time for each case in milliseconds. For compiled JavaScript functions, we record the time across 10 trial runs in 3 major web browsers. For compiled Java functions, we record the time across 10 trial runs under the same inputs.

The baseline "no auto-sanitization" approach overhead is obtained by compiling vanilla templates with no developer's manual sanitizers applied. For our approach, we enable our

CSAS auto-sanitization implementation. To compare the overhead of context-insensitive auto-sanitization, we simply augment all output points with the `escapeHtml` sanitizer during compilation. A direct comparison to Google AutoEscape, the only context-sensitive sanitization approach in templating systems we know of, was not possible because it does not handle rich language features like if-else and loops which create context ambiguities and are pervasive in our benchmarks; a detailed explanation is provided in Section 6.7. To emulate the purely context-sensitive runtime parsing (CSRP) approach, we implemented this technique for our templating langauge. For Java, we directly used an off-the-shelf parser without modifications from the open-source Google AutoEscape implementation in GWT [43]. For JavaScript, since no similar parser was available, we created a parser implementation mirroring the Java-based parser. We believe our implementation was close to the GWT's public implementation for Java, since the overhead is in the same ballpark range.

Results. For JavaScript as the compilation target, the time taken for parsing and rendering the output of all the compiled template output (total 782.584 KB) in 3 major web browsers, averaged over 10 runs, is shown in Figure 6.13. The costs lie between 78% and $4.24x$ for the pure CSRP approach and our approach incurs between $3-9.6\%$ overhead over no sanitization. The primary reason for the difference between our approach and CSRP approach is that the latter requires a parsing of all constant string and context determination of untrusted data at runtime— a large saving in our static type inference approach. Our overhead in JavaScript is due to the application of the sanitizer, which is why our sanitization has nearly the same overhead as the context-insensitive sanitization approach.

For Java, the pure CSRP approach has a 510% overhead, whereas our approach and context-insensitive approach incur no statistically discernable overhead. In summary, our approach achieves the benefits of context-sensitive sanitization at the overhead comparable to a large fraction of other widely used frameworks.

We point out that Closure templates capture the HTML output logic with minimal subsidiary application logic — therefore our benchmarks are heavy in string concatenations and writes to output buffers. As a result, our benchmarks are highly CPU intensive and the runtime costs evaluated here may be amortized in full-blown applications by other latencies (computation of other logic, database accesses, network and file-system operations). For an estimate, XSS-GUARD reports an overhead up to 42% for the CSRP approach [16]. We believe our benchmarks are apt for precisely measuring performance costs of the HTML output logic alone. Further performance optimizations can be achieved for our approach as done in GWT by orthogonal optimizations like caching which mask disk load latencies.

6.7 Related Work

Google AutoEscape, the only other context-sensitive sanitization approach in templating frameworks we are aware of, does not handle the rich language constructs we support— it does not handle conditionals constructs, loops or call operations [42]. It provides safety in straight-line template code for which straight-line parsing and context-determination suffice.

To improve performace, it caches templates and the sanitization requirements for untrusted inputs. Templates can then be included in Java code [43] and C code [42]. As we outline in this chapter, with rich constructs, path-sensitivity becomes a challenging issue and sanitization requirements for untrusted inputs vary from one execution path to the other. AutoEscape's caching optimization does not directly extend to code where sanitization requirements vary depending on executed paths. Our approach, instead, solves the challenges arising from complex language features representative of richer templating systems like Closure Templates.

Context-inference and subsequent context-sensitive placement for .NET legacy applications is proposed in our recent work [97]. The approach proposed therein, though sound, is a per-path analysis and relies on achieving path coverage by dynamic testing. In contrast, the type-based approach in this work achieves full coverage since it is based on static type inference. The performance improvements in our recent dynamic approach relies heavily on the intuition that on most execution paths, developers have manually applied context-sensitive sanitization correctly. The type-based approach in this work can apply sanitization correctly in code completely lacking previous developer-supplied sanitization. A potential drawback of our static approach is that theoretically it may reject benign templates since it reasons about all paths, even those which may be potentially infeasible. In our present evaluation we have not seen such cases.

Analysis techniques for finding scripting vulnerabilities has been widely researched [99, 100, 9, 48, 132, 72, 71, 18, 127, 62, 55, 87, 75, 8]. Defense architectures have targeted three broad categories: server-side techniques [72, 110, 16, 97, 132], purely browser-based techniques [15, 79] and client-server collaborative defenses [61, 84, 49, 105]. Unlike browser-based and client-server defenses, purely server-side approaches are applicable to the server code without requiring modifications to web browsers. Our techniques are an example of this fact.

Among server-side approaches, strong typing has been proposed as a XSS defense mechanism in the work by Robertson et. al [93]. Our approach significantly contrasts theirs in that it does not require any annotations or changes to the existing code, does not rely on strong typing primitives in the base language such as monads and is a mixed static-dynamic type system for existing web templating frameworks and for retrofitting to existing code.

6.8 Conclusion

We present a new auto-sanitization defense to secure web application code from scripting attacks (such as XSS) by construction. We introduce context type qualifiers, a key new abstraction, and develop a type system which is directly applicable to today's commercial templating languages. We have implemented the defense in Google Closure Templates, a state-of-the-art templating system that powers GMail and Google Docs. We find that our mostly static system has low performance overheads, is precise and requires no additional annotations or developer effort. We hope that our abstractions and techniques can be extended

to other complex languages and frameworks in the future towards the goal of eliminating scripting attacks in emerging web applications.

Chapter 7

DSI: A Basis For Sanitization-Free Defense

In the previous chapter, we tackle the problem of auto-sanitization to prevent scripting attacks. However, the underlying issue for scripting attacks is the lack of principled mechanisms in HTML and other web languages to separate trusted code and data from untrusted data embedded inline. As an artifact of this existing design, web developers pervasively use fragile sanitization mechanisms, which have been notoriously hard to get right in the past. In this chapter, we propose a fundamental integrity property, which we term as *document structure integrity*, which can serve as a conceptual basis for defense without requiring any sanitization in web applications. We begin by stating four requirements that a clean-slate solution to scripting attacks should have.

Defense Requirements. Based on these empirical observations of Chapter 5 and Chapter 6, we formulate the following four requirements for a technique to prevent script injection attacks.

1. The defense should not rely on server-side sanitization of untrusted data; instead it should form a second level of defense to safeguard against holes that result from error-prone sanitization mechanism.

2. The defense should confine untrusted data in a manner consistent with the browser implementation and user configuration.

3. The defense must address attacks that target server-side as well as client-side languages.

4. The defense should proactively protect against attacks without relying on detection of common symptoms of malicious activity such as cross-domain sensitive information theft.

Our Approach. In this chapter, we develop an sanitization-free approach. Our approach can be implemented transparently in the web server and the browser requiring minimal

web developer intervention and it provides a second line of defense for preventing scripting attacks. In our approach, script injection vulnerability is viewed as a privilege escalation vulnerability, as opposed to an input sanitization problem. Sanitization and filtering/escaping of untrusted content aims to block or modify the content to prevent it from being interpreted as code. Our approach does not analyze the values of the untrusted data; instead, it restricts the interpretation of untrusted content to certain lexical and syntactic operations. The web developer specifies a restrictive policy for untrusted content, and the web browser enforces the specified policy.

To realize this system we propose a new scheme, which uses markup primitives for the server to securely demarcate inline user-generated data in the web document, and is designed to offer robustness in the face of an adaptive adversary. This allows the web browser to verifiably isolate untrusted data while initially parsing the web page. Subsequently, untrusted data is tracked and isolated as it is processed by higher-order languages such as JavaScript. This ensures the integrity of the document parse tree — we term this property as *document structure integrity* (or DSI). DSI is similar to PreparedStatements [36] which provide query integrity in SQL. DSI is enforced using a fundamental mechanism, which we call *parser-level isolation* (or PLI), that isolates inline untrusted data and forms the basis for uniform runtime enforcement of server-specified syntactic confinement policies.

We discuss the deployment of this scheme in a client-server architecture that can be implemented with a minimum impact to backwards compatibility in modern browsers. Our proposed architecture employs server-side taint tracking proposed by previous research to minimize changes to the web application code. We implemented a proof-of-concept that embodies our approach and evaluated it on a dataset of 5,328 web sites with known scripting vulnerabilities and 500 other popular web sites. Our preliminary evaluation demonstrates that parser-level isolation with a single default policy is sufficient to nullify over 98% of the attacks we studied. Our evaluation also suggests that our techniques can be implemented with very low false positives, in contrast to false positives that are likely to arise due to fixation of policy in purely client-side defenses.

Our work builds on several works which have identified the need for policy-based confinement and isolation of untrusted data [61, 73, 20]. A detailed analytical comparison with previous works is provided in Sections 7.7 and Section 7.9. In comparison to existing scripting defenses, DSI enforcement offers a more comprehensive defense against attacks that extend beyond script injection and sensitive information stealing, and safeguards against both static as well as dynamic integrity threats.

7.1 XSS Definition and Examples

A script injection (or XSS) vulnerability is one that allows injection of untrusted data into a victim web page which is subsequently interpreted in a malicious way by the browser on behalf of the victim web site. This untrusted data could be interpreted as any form of code that is not intended by the server's policy, including scripts and HTML markup. We

```
 1:   <body>
 2:   <div id='WelcomeMess'> Welcome! </div>
 3:   <div id='$GET['FriendID-Status']' name='status'> </div>
 4:   <script>
 5:      if($GET['MainUser'])  {
 6:          document.getElementById('WelcomeMess').innerHTML =
 7:                          "Welcome" + "$GET['MainUser']";
 8:      }
 9:      var divname = document.getElementsByName("status")[0].id;
10:      var Name = divname.split("=")[0]; var Status = divname.split("=")[1];
11:      eval("divname.innerHTML = \"" + Name + " is " + Status + "\"");
12:   </script>
13:   </body>
```

Figure 7.1: Example showing a snippet of HTML pseudocode generated by a vulnerable social networking web site server. Untrusted user data is embedded inline, identified by the `$GET['...']` variables.

Untrusted variable *(Attack Number)*	Attack String
`$GET['FriendID-Status']` *(Attack 1)*	*' onmouseover=javascript:document.location="http://a.com"*
`$GET['MainUser']` *(Attack 2)*	*</script><script>alert(document.cookie);</script>*
`$GET['FriendID-Status']` *(Attack 3)*	*Attacker=Online"; alert(document.cookie);+"*
`$GET['MainUser']` *(Attack 4)*	*<iframe src=http://attacker.com></iframe>*

Figure 7.2: Example attacks for exploiting vulnerabilities in Figure 7.1.

treat only user-generated input as untrusted and use the terms "untrusted data" and "user-generated data" interchangeably in this chapter. We also refer to content as being either *passive*, i.e, consisting of elements derived by language terminals (such as string literals and integers)– or *active*, i.e, code that is interpreted (such as HTML and JavaScript).

An Example. To outline the challenges of preventing exploits for XSS vulnerabilities, we show a toy example of a social networking site in Figure 7.1. The pseudo HTML code is shown here and places where untrusted user data is inlined are denoted by elements of `$GET['...']` array (signifying data directly copied from GET/POST request parameters). In this example, the server expects the value of `$GET['MainUser']` to contain the name of the current user logged into the site, and `$GET['FriendID-Status']` to contain a string with the name of another user and his status message ("online" or "offline") separated by a delimiter ("="). Assuming no sanitization is applied, this code has at least 4 places where

vulnerabilities arise, which we illustrate with possible exploits[1] summarized in Figure 7.2.

- *Attack 1: String split & Attribute injection.* In this attack, the untrusted variable `$GET['FriendID-Status']` could prematurely break out of the `id` attribute of the `<div>` tag on line 3, and inject unintended attributes and/or tags. In this particular instance, the attack string shown in Figure 7.2 closes the string delimited by the single quote character, which allows the attacker to inject the `onmouseover` JavaScript event. The event causes the page to redirect to `http://a.com` potentially fooling the user into trusting the attacker's website.

  A similar attack is possible at line 7, wherein the attacker breaks out of the JavaScript string literal using an end-of-string delimiter (`"`) character in the value for the variable `$GET['MainUser']`.

- *Attack 2: Node splitting.* Even if this server sanitizes `$GET['MainUser']` on line 7 to disallow JavaScript end-of-string delimiters, another attack is possible. The attacker could inject a string to split the enclosing `<script>` environment, and then inject a new `script` tag, as shown by the second attack string in Figure 7.2.

- *Attack 3: Dynamic code injection.* A more subtle attack is one that targets the integrity of the `eval` query on line 11. Notice that JavaScript variable `Name` and `Status` are derived from parsing the untrusted `id` of the `div` element on line 3. Even if the server sanitizes the variable `$GET ['FriendID-Status']` value for use in the `div` element context on line 3 by removing the `'` delimiter, the attacker could still inject code in the dynamically generated javascript `eval` statement. The vulnerability on line 10 parses the `id` attribute value of each div element into separate user name and status variables, which performs no sanitization for variable named `Status`. The attacker can use an attack string value as shown as the third string in Figure 7.2 to execute the arbitrary JavaScript code at line 11.

- *Attack 4: Dynamic active HTML update.* The attacker could inject active elements inside the `<div>` with id `WelcomeMess` at line 6-7, by using the fourth attack string in Figure 7.2 as the value for `$GET['MainUser']`. This attack updates the web page DOM [2] tree dynamically on the client side after the web page has been parsed and the script code has been executed.

Motivation for our approach. We observe that all of the attacks outlined in Figure 7.2 require breaking the intended structure of the parse tree on the browser. The resulting parse trees from all attacks are shown superimposed in Figure 7.3. It is worth noting that attacks 1 and 2 break the structure of the web page during its initial parsing by the HTML and

[1]The sample attacks are illustrative of attacks seen in the past, and are not guaranteed to work on all browsers.

[2]DOM is the parse tree for the HTML code of the web page

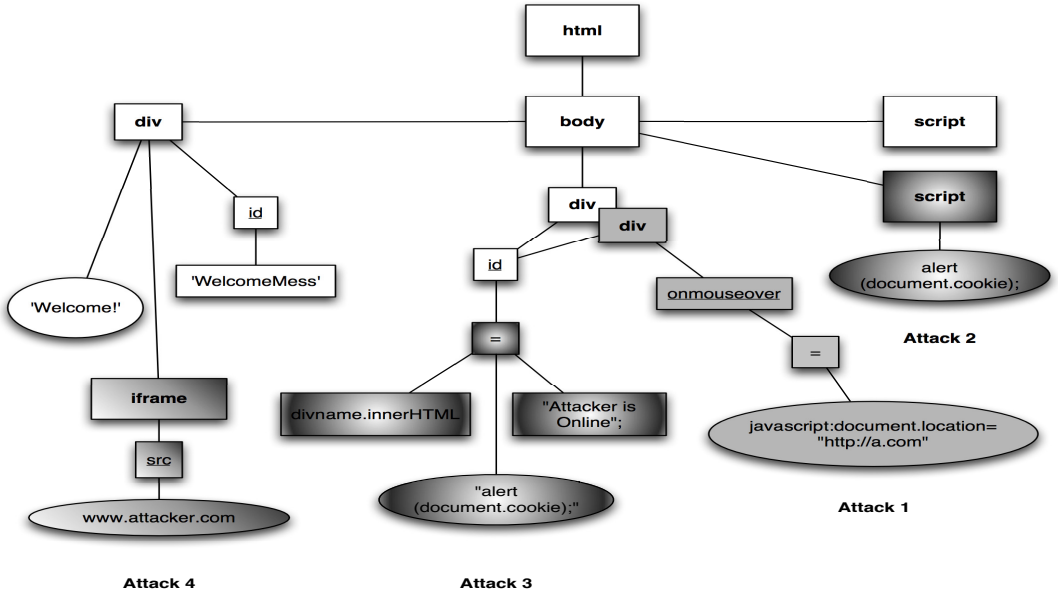

Figure 7.3: Coalesced parse tree for the vulnerable web page in Figure 7.1 showing superimposition of parse trees resulting from all attacks simultaneously. White node show the valid intended nodes whereas the dark nodes show the untrusted nodes inserted by the attacker.

JavaScript parsers, whereas attack 3 and 4 alter the document structure during dynamic client-side operations.

If the browser could robustly isolate untrusted data on the web page, then it can *quarantine* untrusted data with respect to an intended policy. In this example, the server wishes to coerce untrusted nodes to leaf nodes in the parse tree, by treating them as string literals. This disallows injection of any language non-terminal (possible active HTML/JavaScript content) in the web page.

7.2 Approach Overview

Web pages are parsed by various language parsers that are part of the web browser into internal parse trees. Under a benign query, the web server produces a web page that when parsed, results in a parse tree with a certain structure. This parse tree represents the structure that the web server aims to allow in the web document, and hence we term it as the *document structure*. In our approach, we ensure that the browser can identify and isolate nodes derived from user-generated data, in the parse tree during parsing. In principle, we whitelist the intended document structure and prevent the untrusted nodes from changing this structure in unintended ways. We call the property of ensuring intended document structure as enforcing *document structure integrity* or *DSI*.

We clearly separate the notion of a *confinement policy* from the parser-level isolation

mechanism. As in our running example, web sites often wish to restrict untrusted data to leaf nodes in the document structure, as this is an effective way to stop an attacker from injecting active content. We refer to this confinement policy as *terminal confinement*, i.e., confinement of untrusted data to leaves in the document structure, or equivalently, to strings derived from terminals in the grammar representing valid web pages. Figure 7.4 is the parse tree obtained by DSI enforcement for our running example.

The server may wish to instruct the browser to enforce other higher-level semantic policies, such as specifying a restricted sandbox, but this is possible only if the underlying language or application execution framework provides primitives that prevent an attacker from breaking out of the confinement region. For instance, the new proposal of *sandbox* attributes for `iframe` tags (introduced in HTML 5 [119]) defines semantic confinement policies for untrusted data from another domain. However, it relies on the `iframe` abstraction to provide the isolation. Similar to `iframes`, DSI forms the basis for higher level policy specification on web page regions that contain inline untrusted data. Our isolation primitives have no dependence on escaping/quoting or input sanitization for their internal working, thus making our mechanism a strong second line of defense for input validation checks already being used in web application code.

Key challenges in ensuring DSI in web applications. The high-level concept of terminal confinement has been proposed to defend against attacks such as SQL injection [107], but HTML differs from SQL in two significant ways. First, HTML can embed code written in various higher-order languages which share the same inline data. For instance, there are both generic (such as JavaScript URI) and browser-specific ways to invoke functions in VBScript, XUL, JavaScript, CSS and so on. To account for this difficulty, we treat the document structure as that implied by the superimposition of the parse trees obtained from code written in all languages (including HTML, JavaScript) used in a web page.

A second distinguishing challenge in securing web applications, specially AJAX driven applications, is that the document parse trees can be dynamically generated and updated on the client side. In real web pages, code in client-side scripting languages parses web content asynchronously, which results in repeated invocations of different language parsers. To address dynamic parsing, we treat the document structure as having two different components—a static component and a dynamic one. A web page must have a *static document structure*, i.e., the document structure implied by the parse tree obtained from the initial web page markup received by the browser. Similarly, a web page also has a *dynamic document structure*, i.e., the structure implied by the set of parse trees created by different parsers dynamically. To illustrate the distinction, we point out that attacks 1 and 2 in our running example violate static DSI, whereas attacks 3 and 4 violate dynamic DSI.

Goals. Parser-level isolation is a set of mechanisms to ensure robust isolation of untrusted data in the document structure throughout the lifetime of the web application. Using PLI we outline three goals that enforce DSI for a web page with respect to a server-specified policy, say P. First, we aim to enforce *static DSI with respect to* P, from the point the web page is generated by the server to the point at which it is parsed into its initial parse trees

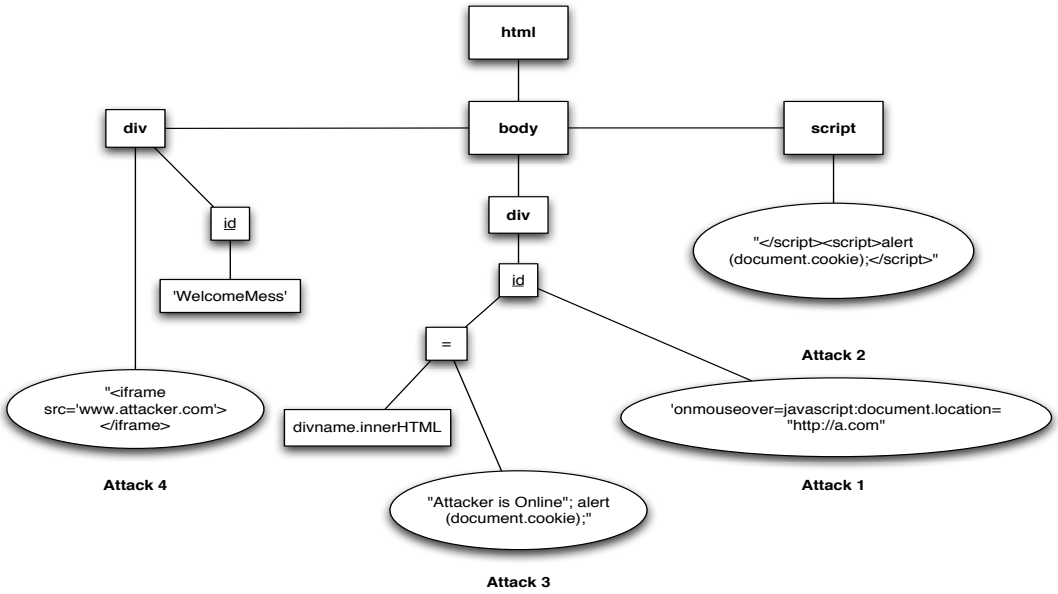

Figure 7.4: Coalesced parse tree (corresponding to parse tree in Figure 7.3) resulting from DSI enforcement with the terminal confinement policy—untrusted subtrees are forced into leaf nodes.

in the browser. As a result, the browser separates untrusted data from trusted data in its initial parse tree robustly. Second, we aim to enforce *dynamic DSI with respect to P* in the browser, across all subsequent parsing operations. Third, we require that the attacker can not evade PLI by embedding untrusted content that results in escalated interpretation of untrusted data. These three goals enforce DSI based on uniform parser-level isolation.

Outline of Mechanisms. We view the operation of encoding the web page in HTML, merely as *serialization* (or marshaling[3]) of the content and the static document structure on the server side, and browser-side parsing of HTML as the deserialization step. We outline 4 steps that implement PLI and ensure the document structure is reconstructed by the browser from the point that the web server generates the web page.

There are two steps that the server implements.

- *Step 1—Separation of trusted and user-generated data.* As a first step, web servers need to identify untrusted data at their output interface, and should distinguish it from trusted application code. We make this assumption to begin with, and discuss some ways to achieve this step through automatic methods in Section 7.4. We believe that this is not an unrealistic assumption—previous work on automatic dynamic taint tracking [132, 87] has shown that tracking untrusted user-generated data at the output interface is possible; in fact, many popular server-side scripting language interpreters (such as PHP) now have built-in support for this. Our goal in subsequent steps is to

[3]akin to serialization in other programming languages and RPC mechanisms

supplement integrity preserving primitives to ensure that the server-specified policy is correctly enforced in the client browser, instead of the sanitization at the server output interface for reasons outlined in the introduction of this chapter.

- *Step 2—Serialization: Enhancement of static structure with markup.* The key to robust serialization is to prevent embedded untrusted data from subverting the mechanism that distinguishes trusted code from inline untrusted data in the browser. To prevent such attacks, we propose the idea of markup randomization, i.e., addition of non-deterministic changes to the markup. This idea is similar to instruction set randomization [64] proposed for preventing traditional vulnerabilities.

There are two steps that the browser implements.

- *Step 3—Deserialization: Browser-side reconstruction of static document structure.* The web browser parses the web page into its initial parse tree, coercing the parse tree to preserve the intended structure. Thus, it can robustly identify untrusted data in the document structure at the end of the deserialization step.

- *Step 4—Browser-side dynamic PLI.* This step is needed to ensure DSI when web pages are dynamically updated. In essence, once untrusted data is identified in the browser at previous step, we initialize it as *quarantined* and track quarantined data in the browser dynamically. Language parsers for HTML and other higher-order languages like JavaScript are modified to disallow quarantined data from being used during parsing in a way that violates the policy. This step removes the burden of having the client-side code explicitly check integrity of the dynamic document structure, as it embeds a reference monitor in the language parsers themselves. Thus, no changes need to be made to existing client-side code for DSI-compliance.

7.3 Enforcement Mechanisms

We describe the high level ideas of the mechanisms in this section. Concrete details for implementing these are described in Section 7.4.

Serialization

Web pages are augmented with additional markup at the server's end, in such a way that the browser can separate trusted structural entities from untrusted data in the static document structure. We call this step serialization, and it is ideally performed at the output interface of the web server.

Adaptive Attacks. One naive way to perform serialization is to selectively demarcate or annotate untrusted data in the web page with special markup. The key concern is that an adaptive attacker can include additional markup to evade the isolation. For instance, let us

```
...
3 :  <div id="⟦5367$GET['FriendId-Status']
⟧5367">
4 :  <script>
5 :  if (⟦3246 $GET['MainUser'] ⟧3246) {
...
```

Figure 7.5: Example of minimal serialization using randomized delimiters for lines 3-5 of the example shown in Figure 7.1.

say that we embed the untrusted data in a contained region with a special tag that disallows script execution that looks like:

```
<div class="noexecute">
    possibly-malicious content
</div>
```

This scheme is proposed in BEEP [61]. As the authors of BEEP pointed out, this naive scheme is weak because an adaptive attacker can prematurely close the `<div>` environment by including a `</div>` in a node splitting attack. The authors of BEEP suggest an alternative mechanism that encodes user data as a JavaScript string, and uses server-side quoting of string data to prevent it from escaping the JavaScript string context. They suggest the following scheme:

```
<div class="noexecute" id="n5"></div>
<script>
 document.getElementById("n5").innerHTML =
 "quoted possibly-malicious content";
</script>
```

We point out that it can be tricky to prevent the malicious content from breaking out of even the simple static JavaScript string context. It is not sufficient to quote the JavaScript end-of-string delimiters (") – an attack string such as `</script><iframe>...</iframe>` perpetrates a node splitting attack closing the script environment altogether, without explicitly breaking out the string context. Sanitization of HTML special characters `<,>` might solve this instance of the problem, but a developer may not employ such a restrictive mechanism if the server's policy allows some form of HTML markup in untrusted data (such as `<p>` or `<b>` tags in user content).

Our goal is to separate the isolation mechanism from the policy. The above outlined attack reiterates that content server-side quoting or validation may vary depending upon the web application's policy and is an error-prone process; keeping the isolation mechanism independent of input validation is an important design goal. We propose the following serialization schemes as an alternative.

Minimal Serialization. In this form of serialization, only the regions of the static web page that contain untrusted data are surrounded by special delimiters. Delimiters are added

around inlined untrusted data independent of the context where the data is embedded. For our running example shown in the Figure 7.1, the serialization step places these delimiters around all occurrences of the `$GET` array variables. If the markup elements used as delimiters are statically fixed, an adaptive attacker could break out of the confinement region by embedding the ending special delimiter in its attack string as discussed above. We propose an alternative mechanism called *markup randomization* (which builds upon our earlier proposal [104]) to defeat such adaptive attacks.

The idea is to generate randomized markup values for special delimiters each time the web page is served, so that the attacker can not deterministically guess the confining context tag it should use to break out. Abstractly, the server appends a integer suffix $c, c \in C$ to a matching pair $[\![\ ]\!]$ of delimiters enclosing an occurrence of untrusted data, to generate $[\![_c\ ]\!]_c$ while serializing. The set C is randomly generated for each web page served. C is sent in a confidential, tamper-proof communication to the browser along with the web page. Clearly, if we use a pseudo-random number generator with a seed C_s to generate C, it is sufficient to send $\{C_s, n\}$, where n is the number of elements in C obtained by repeated invocations of the pseudo-random number generator. In Figure 7.5 , we show the special delimiters added to the lines 3-5 of our running example in Figure 7.1. One recent instance of a minimal serialization scheme is the tag matching scheme proposed in the informal `jail` tag[20], which is analyzed in depth by Louw et. al. [73].

Full Serialization. An alternative to minimal serialization is to mark all trusted structural entities explicitly, which we call full serialization. For markup randomization, the server appends a random suffix $c, c \in C$, to each trusted element (including HTML tags, attributes, values of attributes, strings) and so on.

Though a preferable mechanism from a security standpoint, we need a scheme that can mark trusted elements independent of the context of occurrence with a very fine granularity of specification. For instance, we need mechanism to selectively mark the `id` attribute of the `div` element of line 3 in the running example (shown in Figure 7.1) as trusted (to be able to detect attribute injection attacks), without marking the attribute value as trusted. Only then can we selectively treat the value part as untrusted which can be essential to detect dynamic code injection attacks, such as attack 3 in Figure 7.2.

Independently and concurrent with our work, Gundy et. al. have described a new randomization based full serialization scheme, called Noncespaces [49] that uses XML namespaces. However, XML namespaces does not have the required granularity of specification that is described above, and hence we have not experimented with this scheme. It is possible, however, to apply the full serialization scheme described therein as part of our architecture as well, sacrificing some of the dynamic integrity protection that is only possible with a finer-grained specification. We do not discuss full serialization further, and interested readers are referred to Noncespace [49] for details.

$V \longrightarrow [\![_c N]\!]_c$	$\{N.mark = Untrusted;\}$
$X \longrightarrow Y_1 Y_2$	$\{$`if` $(X.mark == Untrusted)$
	`then` $(Y_1.mark = X.mark;$
	$Y_2.mark = X.mark;)$
	`else` $(Y_1.mark = Trusted;\ \}$
	$Y_2.mark = Trusted;)$

Figure 7.6: Rules for computing `mark` attributes in minimal deserialization.

Deserialization

When the browser receives the serialized web page, it first parses it into the initial static document structure. The document parse tree obtained from deserialization can verifiably identify the untrusted nodes.

Minimal deserialization. Conceptually, to perform deserialization the browser parses as normal, except that it does special processing for randomized delimiters $[\![_c$, $]\!]_c$. It ensures that the token corresponding to $[\![_c$ matches the token corresponding to $]\!]_c$, iff their suffixes are the same random value c and $c \in C$. It also marks the nodes in the parse tree that are delimited by special delimiters as untrusted.

Algorithm to mark untrusted nodes. Minimal deserialization is a syntax-directed translation scheme, which computes an inherited attribute, `mark`, associated with each node in the parse tree, denoting whether the node is `Trusted` or `Untrusted`. For the sake of conceptual explanation, let us assume that we can represent valid web pages that the browser accepts by a context-free grammar G [4].Let $G = \{V, \Sigma, S, P\}$, where V denotes non-terminals, Σ denotes terminals including special delimiters, S is the start symbol, and P is a set of productions. Assuming that C is the set of valid randomized suffix values, the serialized web page s obeys the following rules:

(a) All untrusted data is confined to a subtree rooted at some non-terminal N, such that a production, $V \longrightarrow [\![_c N]\!]_c$, is in P.

(b) Productions of the form $V \longrightarrow [\![_{c_1} N]\!]_{c_2}, c_1 \neq c_2$ are not allowed in P.

(c) $\forall c \in C$, all productions of the form $V \longrightarrow [\![_c N]\!]_c$ are valid in P.

The rules to compute the inherited attribute `mark` are defined in Figure 7.6, with `mark` attribute for S initialized to `Trusted`.

Fail-Safe. Appending random suffixes does not lead to robust design by itself. Sending the set C of random values used in randomizing the additional markups adds robustness against attacker spoofing delimiters.

To see why, suppose C was not explicitly sent in our design. Consider the scenario where an adaptive attacker tries to confuse the parser by generating two valid parse trees. In Figure 7.7 the attacker embeds delimiter $[\![_{2222}$ in `$GET['FriendId-Status']` and a matching delimiter $]\!]_{2222}$ in `$GET['MainUser']`. There could be two valid parse trees—one that matches

[4]practical implementations may not strictly parse context-free grammars

```
...
3 :  <div id="⟦5367..  ⟦2222...
⟧5367">
4 :  <script>
5 :  if (⟦3246 ..  ⟧2222...  ⟧3246) {
...
```

Figure 7.7: One possible attack on minimal serialization, if C were not explicitly sent. The attacker provides delimiters with the suffix 2222 to produce 2 valid parse trees in the browser.

delimiters with suffix 5367 and 3246, and another that matches the delimiters with suffix 2222. Although, the browser could allow the former to be selected as valid as delimiter with 5367 is seen first earlier in the parsing, this is a fragile design because it relies on the server's ability to inject the constraining tag first and requires sequential parsing of the web page. In practice, we can even expect the delimiter placement may be imperfect or missing in cases. For instance in Figure 7.7, if the special delimiters with suffix 5367 were missing, then even if the server had sanitized `$GET['FriendId-Status']` perfectly against string splitting attack (attack 1 in Section 7.1), the attacker possesses an avenue to inject a spurious delimiter tag $⟦_{2222}$. All subsequent tags placed by the server would be discarded in an attempt to match the attacker provided delimiter. The attacker's ability to inject isolation markup is a weakness in the mechanism which does not explicitly send C. The informal `<jail>` proposal may be susceptible to such attacks as well [20]. Our explicit communication of C alleviates this concern.

Browser-side dynamic PLI

Once data is marked untrusted, we initialize it as quarantined. With each character we associate a *quarantine bit*, signifying whether it is quarantined or not. We dynamically track quarantined metadata in the browser. Whenever the base type of the data is converted from the data type in one language to a data type in another, we preserve the quarantine bit through the type transformation. For instance, when the JavaScript code reads a string from the browser DOM into a JavaScript string, the appropriate quarantine bit is preserved. Similarly, when a JavaScript string is written back to a DOM property, the corresponding HTML lexical entities preserve the dynamic quarantine bit.

Quarantine bits are updated to reflect data dependences between higher-order language variables, i.e. for arithmetic and data operations (including string manipulation), the destination variable is marked quarantined, iff any source operand is marked quarantined. We do not track control dependence code as we do not consider this a significant avenue of attack in benign application. We do summarize quarantine bit updates for certain functions which result in data assignment operations but may internally use table lookups or control dependence in the interpreter implementation to perform assignments. For instance, the JavaScript `String.fromCharCode` function requires special processing, since it may use conditional switch statement or a table-lookup to convert the parameter bytes to a string

elements. In this way, all invocations of the parsers track quarantined data and preserve this across data structures representing various parse trees.

Example. For instance, consider the attack 3 in our example. It constructs a parse tree for the `eval` statement as shown in Figure 7.3. The initial string representing the terminal `id` on line 3 is marked quarantined by the deserialization step. With our dynamic quarantine bit tracking, the JavaScript internal representation of the div's `id` and variables `divname`, `Name` and `Status` are marked quarantined. According to the terminal confinement policy, during parsing our mechanism detects that the variable `Status` contains a delimiter non-terminal "`;`". It coerces the lexeme " `;`" to be treated a terminal character rather than interpreting it as a separator non-terminal, thus nullifying the attack.

7.4 Architecture

In this section, we discuss the details of a client/server architecture that embodies our approach. We first outline the goals we aim to achieve in our architecture and then outline how we realize the different steps proposed in Section 7.3.

Architecture Goals

We propose a client-server architecture to realize DSI. We outline the following goals for web sites employing DSI enforcement, which are most important to make our approach amenable for adoption in practice.

1. *Render in non-compliant*[5] *browsers, with minimal impact.* At least the trusted part of the document should render as original in non-compliant browsers. Most user-generated data is benign, so even inlined untrusted data should render with minimal impact in non-compliant browsers.

2. *Low false positives.* DSI-compliant browsers should raise very few or no false positives. A client-server architecture, such as ours, reduces the likelihood of false positives that arise from a purely-client side implementation of DSI (see Section 7.6).

3. *Require minimal web application developer effort.* Automated tools should be employed to retrofit DSI mechanisms to current web sites, without requiring a huge developer involvement.

Client-Server Co-operation Architecture

Identification of Untrusted data. Manual code refactoring is possible for several web sites. Several web mashup components, such as Google Maps, separate the template code of

[5]Web browsers that are not DSI-compliant are referred to as *non-compliant*

the web application from the untrusted data already, but rely on sanitization to prevent DSI attacks. Our explicit mechanisms would make this distinction easier to specify and enforce.

Automatic transformation to enhance the markup generated by the server is also feasible for several commercial web sites. Several server side dynamic and static taint-tracking mechanisms [132, 70, 117] have been developed in the past. Languages such as PHP, that are most popularly used, have been augmented to dynamically track untrusted data with moderate performance overheads, both using automatic source code transformation [132] as well as manual source code upgrades for PHPTaint [117]. Automatic mechanisms that provide taint information could be directly used to selectively place delimiters at the server output.

We have experimented with PHPTaint [117], an implementation of taint-tracking in the PHP 5.2.5 engine, to automatically augment the minimal serialization primitives for all tainted data seen in the output of the web server. We enable dynamic taint tracking of GET/POST request parameters and database pulls. We disable taint declassification of data when sanitized by PHP sanitization functions (since we wish to treat even sanitized data as potentially malicious). All output tainted data are augmented with surrounding delimiters for minimal serialization. Our modifications shows that automatic serialization is possible using off-the-shelf tools.

For more complex web sites that use a multi-component architecture, cross-component dynamic taint analysis may be needed. This is an active area of research and automatic support for minimal serialization at the server side would readily benefit from advances in this area. Recent techniques proposed for program analysis to identify taint-style vulnerabilities [75, 62] could help identify taint sink points in larger web application, where manual identification is hard. Similarly, Nanda et al. have recently shown cross-component dynamic taint tracking for the LAMP architecture is possible [85].

Communicating valid suffixes. In our design it is sufficient to communicate $\{C_s, n\}$ in a secure way, where C_s is the random number generator seed to use and n is the number of invocations to generate the set C of valid delimiter suffixes. Our scheme communicates these as two special HTML tag attributes, (`seed` and `suffixsetlength`), as part of the HTML `head` tag of the web page. We assume that the server and the browser use the same implementation of the psuedo-random number generator. Once read by the browser, it generates this set for the entire lifetime of the page and does not recompute it even if the attacker corrupts the value of the special attributes dynamically. We have verified that this scheme is backwards compatible with HTML handling in current browsers, i.e, these special attributes are completely ignored for rendering in current browsers[6].

Choice of serialization alphabet for encoding delimiters. We discuss two schemes for encoding delimiters.

- We propose use of byte values from the Unicode Character Database [114] which are

[6] "current browsers" refers to: Safari, Firefox 2/3, Internet Explorer 6/7/8, Google Chrome, Opera 9.6 and Konqueror 3.5.9 in this chapter.

rendered as whitespace on the major browsers independent of the selected character set used for web page decoding. Our rationale for using whitespace characters is its uniformity across all common character sets, and the fact that this does not hinder parsing of HTML or script in most relevant contexts (including between tags, between attributes and values and strings). In certain exceptional contexts where these may hinder semantics of parsing, these errors would show up in pre-deployment testing and can easily be fixed. There are 20 such character values which can be used to encode start and end delimiter symbols. All of these characters render as whitespace on cuurent browsers. To encode the delimiters' random suffixes we could use the remaining 18 (2 are used for delimiters themselves) as symbols. Thus, each symbol can encode 18 possible values, so a suffix $\ell - symbols$ long, should be sufficient to yield an entropy of $\ell \times (lg(18))$ or $(\ell \times 4.16)$ bits.

It should be clear that a compliant browser can easily distinguish pages served from a non-compliant web server vs. a randomization compliant web server—it looks at the `seed` attribute in the `<head>` element of the web page. When a compliant browser views a non-compliant page, it simply treats the delimiter encoding bytes as whitespace as per current semantics, as this is a non-compliant web page. When a compliant browser renders a compliant web page, it treats any found delimiter characters as valid iff they have valid suffixes, or else it discards the sequence of characters as whitespace (these may occur by chance in the original web page, or may be attacker's spoofing attempts). Having initialized the enclosed characters as untrusted in its internal representation, it strips these whitespace characters away. Thus, the scheme is secure whether the page is DSI-compliant or not.

- Another approach is to use special delimiter tags, `<qtag>`, by introducing an attribute `check=`*`suffix`*, as well. Qtags have a lesser impact on readability of code than the above scheme. Qtags have the same encoding mechanism as `<jail>` tags proposed informally [20]. We verified that it renders safely in today's popular browsers in most contexts, but is unsuitable to be used in certain contexts such as within strings. Another issue with this scheme is that XHTML does not allow attributes in end tags, and so they don't render well in XHTML pages on non-compliant browsers, and may be difficult to set accepted as a standard.

Policy Specification. Our policies confine untrusted data only. Currently, we support per-page policies that are enforced for the entire web page, rather than varying region-based policies. By default, we enforce the terminal confinement policy which is a default fail-close policy. In most cases, this policy is sufficient for several web sites to defend against reflected XSS attacks. A more flexible policy that is useful is to allow certain HTML syntactic constructs in inline untrusted data, such as restricted set of HTML markup in user blog posts. We support a whitelist of syntactic HTML elements as part of a configurable policy.

We allow configurable specification of whitelisted HTML construct names through a `allowuser` tag attribute for the HTML `<meta>` tag. It can have a comma-separated list of

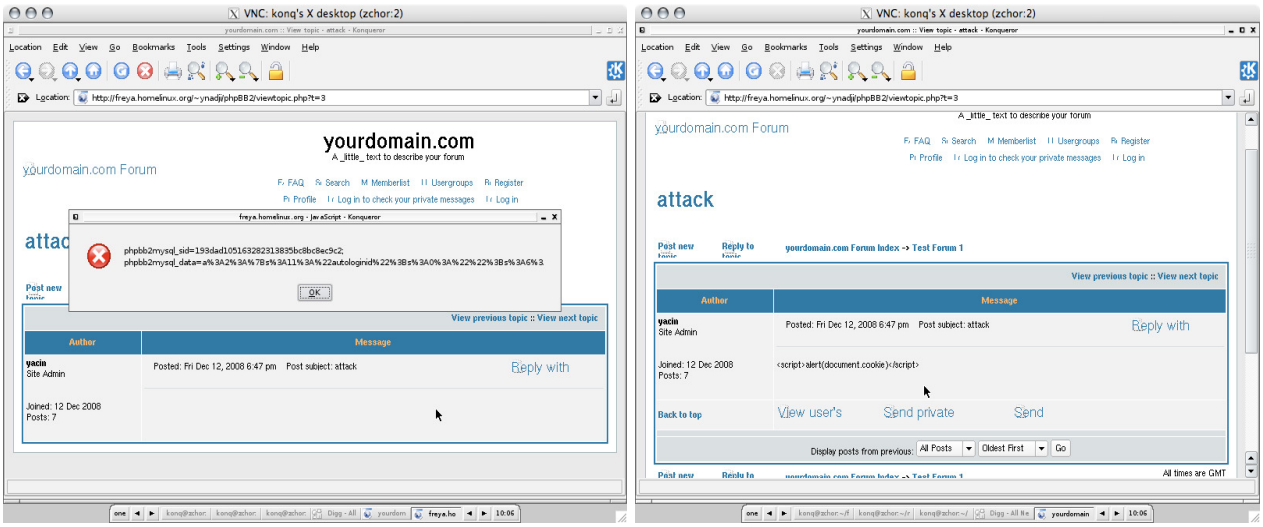

Figure 7.8: (a) A sample web forum application running on a vulnerable version of phpBB 2.0.18, victimized by stored XSS attack as it shows with vanilla Konqueror browser (b) Attack neutralized by our proof-of-concept prototype client-server DSI enforcement.

allowed tags. For instance, the following specification would allow untrusted nodes corresponding to the paragraph, boldface, line break elements, the attribute `id` (in all elements) and the anchor element with optional `href` attribute (only with anchor element) in parse tree to not be flagged as an exploit. The markup `<meta allowuser='p,b,br,@id,a@href'>` renders properly in non-compliant browsers since unknown markup is discarded in the popular browsers.

For security, untrusted data is disallowed to define `allowuser` tag without exception. Policy development and standardization of default policies are important problems which involve a detail study of common elements that are safe to allow on most web sites. However, we consider this beyond the scope of this chapter, but deem worthy of future work.

7.5 Implementation

We discuss details of our prototype implementation of a PLI enabled web browser and a PLI enabled web server first. Next, we demonstrate an example forum application that was deployed on this framework requiring no changes to application code. Finally, we outline the implementation of a web proxy server used for evaluation in section 7.6.

DSI compliant browser. We have implemented a proof-of-concept PLI enabled web browser by modifying Konqueror 3.5.9. Before each HTML parsing operation, the HTML parsing engine identifies special delimiter tags. This step is performed before any character decoding is performed, and our choice of unicode alphabet for delimiters ensures that we deal with all character set encodings. The modified browser simulates a pushdown automaton

during parsing to keep track of delimiter symbols for matching. Delimited characters are initialized as quarantined, which is represented by enhancing the type declaration for the character class in Konqueror with a quarantine bit. Parse tree nodes that are derived from quarantined characters are marked quarantined as well. Before any quarantined internal node is updated to the document's parse tree, the parser invokes the policy checker which ensures that the parse tree update is permitted by the policy. Any internal nodes that are not permitted by the policy are collapsed with their subtree to be treated as a leaf node and rendered as a string literal.

We modified the JavaScript interpreter in Konqueror 3.5.9 to facilitate automatic quarantine bit tracking and prevented tainted access through the JavaScript-DOM interface. The modifications required were a substantial implementation effort compared to the HTML parser modifications. Internal object representations were enhanced to store the quarantine bits and handlers for each JavaScript operation had to be altered to propagate the quarantine bits. The implemented policy checks ensure that quarantined data is only interpreted as a terminal in the JavaScript language.

DSI compliant server. We employed PHPTaint [117] which is an existing implementation dynamic taint tracking in the PHP interpreter. It enables taint variables in PHP and can be configured to indicate which sources of data are marked tainted in the server. We made minor modifications to PHPTaint to integrate in our framework. By default when untrusted data is processed by a built-in sanitization routine, PHPTaint endorses the data as safe and *declassifies*(or clears) the taint; we changed this behavior to not declassify taint in such situations even though the data is sanitized. Whenever data is echoed to the output we interpose in PHPTaint and surround tainted data with special delimiter tags with randomized values at runtime. For serialization, we used the unicode characters `U+2029` as a start-delimiter. Immediately following the start-delimiter are ℓ randomly chosen unicode whitespace characters, the *key*, from the remaining 18 unicode characters. We have chosen $\ell = 10$, though this is easily configurable in our implementation. Following the key is the end-delimiter `U+2028` to signify the key has been fully read.

Example application. Figure 7.8(a) shows a vulnerable web forum application, phpBB version 2.0.18, running on a vanilla Apache 1.3.41 web server with PHP 5.2.5 when viewed with a vanilla Konqueror 3.5.9 with no DSI enforcement. The attacker posts a post containing a script tag which results in a cookie alert. To prevent such attacks, we deployed the phpBB forum application on our DSI-compliant web server next. We required *no* changes to the web application code to deploy it on our prototype DSI-compliant web server. Figure 7.8(b) shows how the attack is nullified by our client-server DSI enforcement prototype which employs PHPTaint to automatically mark forum data (derived from the database) as tainted, enhances it with minimal serialization which enables a DSI-compliant version of Konqueror 3.5.9 to nullify the attack.

Client-side Proxy Server. For evaluation of the 5,328 real-world web sites, we could not use our prototype taint-enabled PHP based server because we do not have access to server code of the vulnerable web sites. To overcome this practical limitation, we implemented a

client-side proxy server that approximately mimics the server-side operations.

When the browser visits a vulnerable web site, the proxy web server records all GET/-POST data sent by the browser, and maintains state about the HTTP request parameters sent. The proxy essentially performs *content based tainting* across data sent to the real server and the received response, to approximate what the server would do in the full deployment of the client-server architecture.

The web server proxy performs a lexical string match between the sent parameter data and the data it receives in the HTTP response. For all data in the HTTP response that matches, the proxy performs minimal serialization (approximating the operations of a DSI-compliant server) i.e, it lexically adds randomized delimiters to demarcate matched data in the response page as untrusted, before forwarding it to the PLI enabled browser.

7.6 Evaluation

To evaluate the effectiveness and overhead of PLI and PLI enabled browsers we conducted experiments with two configurations. The first configuration consists of running our prototype PLI enabled browser and a server running PHPTaint with the phpBB application. This configuration was used to evaluate effectiveness against stored XSS attacks. The second configuration ran our PLI enabled web browser directing all HTTP requests to the proxy web server described in section 7.6. The second configuration was used to study real-world reflected attacks, since we did not have access to the vulnerable web server code.

Experimental Setup

Our experiments were performed on two systems—one ran a Mac OS X 10.4.11 on a 2.0 GHz Intel processor with 2GB of memory, and the other runs Gentoo GNU/Linux 2.6.17.6 on a 3.4 GHz Intel Xeon processor with 2 GB of memory. The first machine ran an Apache 1.3.41 web server with PHP 5.2.5 engine and MySQL back-end, while the second ran the DSI compliant Konqueror. The two machines were connected by a 100 Mbps switch. We configured our prototype PLI enabled browser and server to apply the default policy of terminal confinement to all web requests unless the server overrides with another whitelisting based policy.

Attack Detection

We measure the effectiveness of our prototype implementation against reflected XSS and stored XSS attacks.

Reflected XSS. We evaluated the effectiveness against all real-world web sites with known vulnerabilities, archived at the XSSed [130] web site as of 25th July 2008, which resulted in successful attacks using Konqueror 3.5.9. In this category, there were 5,328 web sites which constituted our final test dataset. Our DSI-enforcement using the proxy web server and DSI compliant browser nullified 98.4% of these attacks as shown in Figure 7.9. Upon further

Attack Category	*# Attacks*	*# Prevented*
Reflected XSS	5,328	5,243 (98.4%)
Stored XSS	25	25 (100%)

Figure 7.9: Effectiveness of DSI enforcement against both reflected XSS attacks [130] as well as stored XSS attack vectors [94].

analysis of the false negatives in this experiment, we discovered that 46 of the remaining cases were missed because the real web server modified the attack input before embedding it on the web page—our web server proxy failed to recognize this server-side modification as it performs a simple string matching between data sent by the browser and the received HTTP response. We believe that in a full deployment these would be captured by server-side taint tracking. We could not determine the cause of missing the remaining 39, as the sent input was not discernible in the HTTP response web page. Overall, this shows that the policy of terminal confinement, if supported in web servers as the default, is sufficient to prevent a large majority of reflected XSS attacks.

Stored XSS. We setup a vulnerable version of the phpBB web blog application (version 2.0.18) on our DSI enabled web server, and injected 30 benign text and HTML based posts, and all of the stored attack vectors taken from XSS cheat sheet [94] that worked in Konqueror 3.5.9. Of the 92 attack vectors outlined therein, only 25 worked in a vanilla Konqueror 3.5.9 browser. We configured the policy to allow only `<p>, <b> and <a>` HTML tags and `href` attributes. No modifications were made to the phpBB application code. Our prototype nullified all 25 XSS attacks.

Performance

We estimate the performance overheads of a DSI-compliant web browser and web server.

Browser Performance. To measure the browser performance overhead, we compared the page load times of our modified version of Konqueror 3.5.9 and the vanilla version of Konqueror 3.5.9. We evaluated against the test benchmark internally used at Mozilla for browser performance testing, consisting of over 350 web pages of popular web pages with common features including HTML, JavaScript, CSS, and images[83]. No data on this web pages was marked untrusted. We measured a performance overhead of 1.8% averaged over 5 runs of the benchmark.

We also measured the performance of loading all the pages from the XSSed dataset consisting of 5,328, with untrusted data marked with serialization delimiters. We observed a similar overhead of 1.85% when processing web pages with tainted data.

Web page (or code) size increase often translates to increased corporate bandwidth consumption, and is important to characterize in a cost analysis. For the XSSed dataset, our instrumentation with delimiters of length $\ell = 10$ increased the page size by less than 1.1% on average for all the web pages with marked untrusted data.

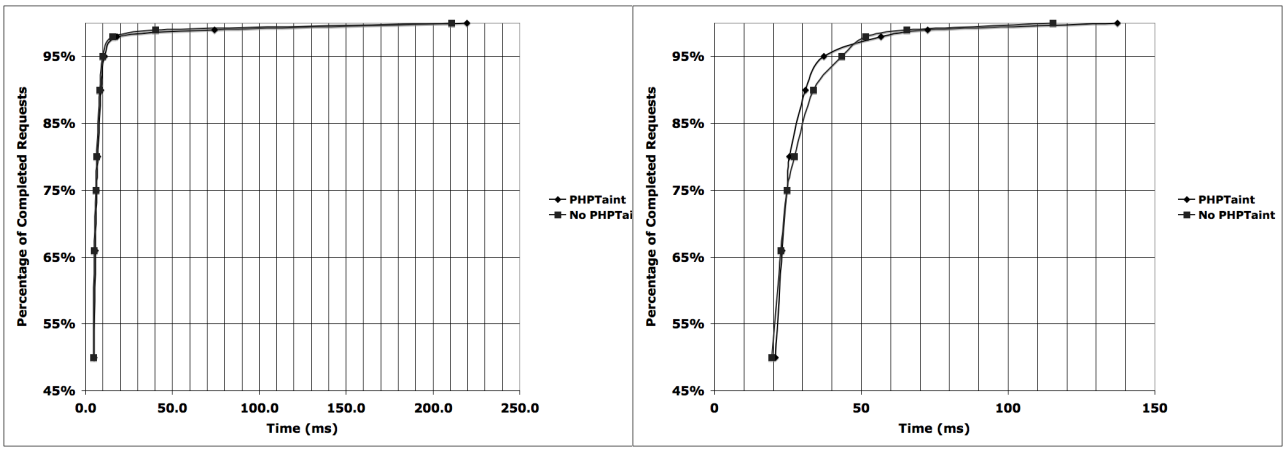

Figure 7.10: Percentage of responses completed within a certain timeframe. 1000 requests on a 10 KB document with (a) 10 concurrent requests and (b) 30 concurrent requests.

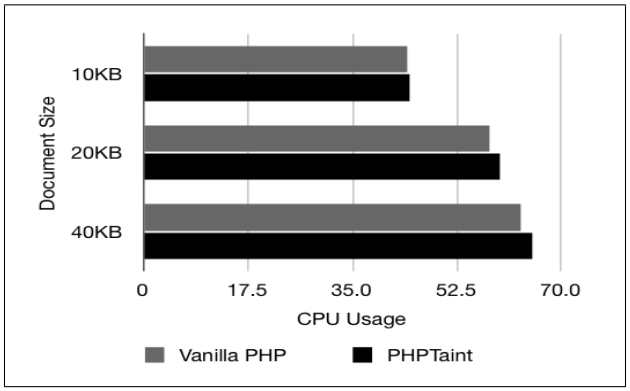

Figure 7.11: Increase in CPU overhead averaged over 5 runs for different page sizes for a DSI-enabled web server using PHPTaint [117].

Server Performance. We measured the CPU overhead for the phpBB application running on a DSI compliant web server with PHPTaint enabled. This was done with `ab` (ApacheBench), a tool provided with Apache to measure performance [2]. It is configured to generate dynamic forum web pages of sizes varying from 10 KB to 40 KB. In our experiment, 64,000 requests were issued to the server with 16 concurrent requests. As shown in Figure 7.11, we observed average CPU overheads of 1.2%, 2.9% and 3.1% for pages of 10 KB, 20 KB, and 40 KB in size respectively. This is consistent with the performance overheads reported by the authors of PHPTaint [117]. Figure 7.10 shows a comparison between the vanilla web server and a DSI-compliant web server (both running phpBB) in terms of the percentage of HTTP requests completed within a certain response time frame. For 10 concurrent requests, the two servers perform very similarly, whereas for 30 concurrent requests the server with PHPTaint shows some degradation for completing more than 95% of the

requests.

False Positives

We observed a lower false positive rate in our stored XSS attacks experiment than in the reflected XSS experiment. In the stored experiment, we did not observe any false positives. In the reflected XSS experiment, we observed false positives when we deliberately provided inputs that matched existing page content. For the latter experiment, we manually browsed the Global Top 500 websites listed on Alexa [4] browsing with deliberate intent to raise false positives. For each website, we visited an average of 3 second-level pages by creating accounts, logging in with malicious inputs, performing searches for dangerous keywords, as well as clicking on links on the web pages to simulate normal user activity.

With our default policy, as expected, we were able to induce false positives on 5 of the web pages. For instance, a search query for the string "`<title>`" on Slashdot[7] caused benign data on the returned page to be marked quarantined. We confirmed that these arise because our client-side proxy server marks trusted code as untrusted which subsequently raises alarms when interpreted as code by the browser. In principle, we expect that full implementation with a taint-aware server side component would eliminate these false positives inherent in the client-side proxy server approximation.

We also report that even with the client-side proxy server approximation, we did *not* raise false positives in certain cases where the IE 8 Beta XSS filter did. For instance, we do not raise false positives when searching for the string "javascript:" on Google search engine. This is because our DSI enforcement is parser context aware—though all occurrences of "javascript:" are marked untrusted in the HTTP response page, our browser did not raise an alert as untrusted data was not interpreted as code.

7.7 Comparison with Existing XSS Defenses

We outline the criteria for analytically comparing different XSS defenses first, and then discuss each of the existing defenses next providing a summary of the comparison in Figure 7.12.

Comparison Criteria. To concretely summarize the strengths and weaknesses of various XSS defense techniques, we present a defender-centric taxonomy of adaptive attacks to characterize the ability of current defenses against current attacks as well as attacks in the future that try to evade the defenses. Adaptive attackers can potentially target at least the avenues outlined below.

- *Browser inconsistency.* Inconsistency in assumptions made by the server and client lead to various attacks as outlined earlier.

[7] http://slashdot.org

- *Lexical Polymorphism.* To evade lexical sanitization, attackers may find variants in lexical entities.
- *Keyword Polymorphism.* To evade keyword filters, attackers may find different syntactic constructs to bypass these. For instance, in the Samy worm [96], to inject a restricted keyword `innerHTML`, the attacker used a semantically equivalent construct " `eval ('inner'+'HTML')`".
- *Multiple Injection Vectors.* Attacker can inject non-script based elements.
- *Breaking static structural integrity.* To specifically evade confinement based schemes, attacker can break out of the static confinement regions on the web page.
- *Breaking dynamic structural integrity.* Attacks may target breaking the structure of the dynamically executing client-side code, as discussed in Section 7.1.

Defense against each of the above adaptive attack categories serves a point of comparing existing defenses. In addition to these, we analytically compare the potential effectiveness of techniques to defend against stored XSS attacks. We also characterize whether a defense mechanism enables flexible server-side specification of policies or not. This is important because fixation of policies often results in false positives, especially for content-rich untrusted data, which can be a serious impediment to the eventual deployability of an approach.

Figure 7.12 shows the comparative capabilities of existing defense techniques at a glance on the basis of criteria outlined earlier in this section. We describe current XSS defenses and discuss some of their weaknesses.

Purely server-side defenses

Input Validation and sanitization. Popular server side languages such as PHP provide standard sanitization functions, such as `htmlspecialchars`. However, the code logic to check validity is often concentrated at the input interface of the server, and also distributed based on the context where untrusted data gets embedded. This mechanism serves as a first line of defense in practice, but is not robust as it places excessive burden on the web developer for its correctness. The prevalence of XSS attacks today shows that these mechanisms fail to safeguard against both static and dynamic DSI attacks.

Browser-independent Policy Checking at Output. Taint-tracking [132, 85, 87, 90] on the server-side aims to centralize sanitization checks at the output interface with the use of taint metadata. Since the context of where untrusted data are being embedded can be arbitrary, the policy checking becomes complicated especially when dealing with attacks that affect dynamic DSI. The primary reason is the lack of semantics of client side behavior in the policy checking engine at the interface. Another problem with this approach is that the policy checks are not specific to the browser that the client uses and can be susceptible to browser-server inconsistency bugs.

Techniques	BI	P	MV	S DSI	D DSI	ST	FP
Purely Server-side							
Input Validation & Sanitization			✓			✓	✓
Server Output browser-independent policies (using taint-tracking)		✓	✓	✓		✓	✓
Server Output Validation browser-based policies (XSS-GUARD [16])		✓	✓	✓	✓	✓	✓
Purely Browser Side							
Sensitive Information Flow Tracking	✓	✓		✓	✓	✓	
Global Script Disabling	✓	✓		✓	✓	✓	
Personal Firewalls with URL Blocking	✓	✓			✓		
GET/POST Request content based URL blocking	✓	✓		✓		✓	
Browser-Server Cooperation Based							
Script Content Whitelisting (BEEP)	✓	✓		✓		✓	✓
Region Confinement Script Disabling (BEEP)	✓	✓		✓		✓	✓
PLI with Server-specified policy enforcement	✓	✓	✓	✓	✓	✓	✓

BI	Not susceptible to browser-server inconsistency bugs
P	Designed to easily defeats lexical and keyword polymorphism based attacks
MV	Designed for comprehensiveness against multiple vectors and attack goals (Flash objects as scripting vectors, `iframe`s insertion for phishing, click fraud).
S DSI	Designed to easily defeat evasion attacks that break static DSI (attacks such as 1,2 in Section 7.1).
D DSI	Designed to easily defeat evasion attacks that break dynamic DSI (attacks such as 3,4 in Section 7.1).
ST	Can potentially deal with stored XSS attacks.
FP	Allows flexible server configurable policies (important to eliminate false positives for content-rich untrusted data)

Figure 7.12: Various XSS Mitigation Techniques Capabilities at a glance. Columns 2 - 6 represent security properties, and columns 7-9 represent other practical issues. A '✓' denotes that the mechanism demonstrates the property.

Browser-based Policy Checking at Output. To mitigate the lack of client-side language semantics at the server output interface, XSS-GUARD [16] employs a complete browser implementation on the server output. In principle, this enables XSS-GUARD to deal with both static and dynamic DSI attacks, at the expense of significant performance overheads. However, this scheme conceptually still suffers from browser inconsistency bugs as a different target browser may be used by the client than the one checked against. Our technique enables the primary benefits of XSS-GUARD without high performance overheads and making the policy enforcement consistent with the client browser.

Purely client-side defenses

Sensitive information flow tracking. Vogt et. al. propose sensitive information flow tracking [118] in the browser to identify spurious cross-domain sensitive information transfer as a XSS attack. This approach is symptom targeted and limited in its goal, and hence does not lend easily to other attack targets outlined in the introduction. It also requires moderately high false positives in normal usage. This stems from the lack of specification of the intended policy by the web server.

Script Injection Blocking. Several techniques are focused on stopping script injection attacks. For instance, the Firefox NoScript extension block scripts globally on web sites the user does not explicitly state as trusted. Many web sites do not render well with this extension turned on, and this requires user intervention. Once allowed, all scripts (including those from attacks) can run in the browser.

Personal Firewalls with URL blocking. Noxes [67] is a client-side rule based proxy to disallow users visiting potentially unsafe URL using heuristics. First, such solutions are not designed to distinguish trusted data generated by the server from user-generated data. As a result, they can have high false negatives (Noxes treats static links in the page as safe) and have false positives [67] due to lack of server-side configuration of policy to be enforced. Second, they are largely targeted towards sensitive information stealing attacks.

GET/POST Request content based URL blocking. Several proposals aim to augment the web browser (or a local proxy) to block URLs that contain GET/POST data with known attack characters or patterns. The most recent is an implementation of this is the XSS filter in Internet Explorer (IE) 8 Beta [56]. First, from our limited experiments with the current implementation, this approach does not seem to detect XSS attacks based on the parsing context. This raises numerous false positives, one instance of which we describe in Section 7.6. Second, their design does not allow configurable server specified policies, which may disallow content-rich untrusted data. In general, fixed policies on the client-side with no server-side specification either raise false positives or tend to be too specific to certain attack vectors (thus resulting in false negatives). Finally, our preliminary investigation reveals that they currently do not defend against integrity attacks, as they allow certain non-script based attack vectors (such as forms) to be injected in the web page. We believe this is an interesting avenue and a detailed study of the IE 8 mechanism would be worthwhile to

understand capabilities of such defenses completely.

Client-server cooperative defenses

This paradigm for XSS defense has emerged to deal with the inefficiencies of purely client and server based mechanisms. Jim et al. have recently proposed two approaches in BEEP [61]—whitelisting legitimate scripts and defining regions that should not contain any scripting code.

Whitelisting of legitimate scripts. First, BEEP targets only script-injection based vectors and hence is not designed to comprehensively defend against other XSS vectors. Second, BEEP's mechanism does not thwart attacks (such as attack 4 in Figure 7.2) violating dynamic DSI that target unsafe usage of data by client-side code. Their mechanism checks the integrity and authenticity of the script code before it executes, but does not directly extend to attacks that deal with the safety of data usage. Our technique enforces a dynamic parser-level confinement to ensure that data is not interpreted as code in client-side scripting code. Follow-on proposals such as Content-Security Policy (CSP) do address some of these shortcomings [105]; we refer the readers to work by Weinberger et. al. for challenges in implementation of CSP in real web sites [124].

Region-based Script Disabling. BEEP outlined a technique to define regions of the web page that can not contain script code, which allows finer-grained region-based script disabling than those possible by already supported browser mechanisms [88]. First, their isolation mechanism uses JavaScript string quoting to prevent static DSI attacks against itself. As discussed in Section 7.3, this mechanism can be somewhat tricky to enforce for content-rich untrusted data which allows HTML entities in untrusted data. Second, this mechanism does not deal with dynamic DSI attacks by itself, because region based script blocking can not be applied to script code regions.

7.8 Discussion

DSI enforcement using a client-server architecture offers a strong basis for XSS defense in principle. However, we discuss some practical concerns for a full deployment of this scheme. First, our approach requires both client and server participation in implementing our enhancements. Though we can minimize the developer effort for such changes, our technique requires both web servers and clients to collectively upgrade to enable any protection.

Second, a DSI-compliant browser requires quarantine bit tracking across operations of several languages. If implemented for JavaScript, this would prevent attacks vectors using JavaScript, but not against attacks that using other languages. Uniform cross-component quarantine bit tracking is possible in practice, but it would require vendors of multiple popular third party web plugins (Flash, Flex, Silverlight, and so on) to cooperate and enhance their language interpreters or parsers. Automatic techniques to facilitate such propagation and cross-component dynamic quarantine bit propagation at the binary level for DSI

enforcement are interesting research directions for future work that may help address this concern.

Third, it is important to account for end-user usability. Our techniques aim to minimize the impact of rendering DSI compliant web pages on existing web browsers for ease of transition to DSI compliance; however, investigation of schemes that integrate DSI seamlessly while ensuring static DSI are important. Louw et. al. identify the need for isolation of untrusted content in static HTML markup [73]; they present a detailed comparison of prevalent isolation mechanisms in HTML. In our work, we outline techniques that address static as well as dynamic isolation of untrusted data and discuss how these mechanisms can be used in conjunction with other techniques to robustly thwart XSS attacks. We hope that our analysis of the primitives from an adaptive attacker's perspective. Our exposition of XSS as a static-dynamic integrity violation provide additional insight for development of newer language primitives for isolation. Finally, we recognise that false positives are another concern for usability. We did not encounter false positives in our preliminary evaluation and testing, but we believe that this is not sufficient to rule out its possibility in a full deployment of this scheme.

7.9 Related Work

XSS defense techniques can be largely classified into detection techniques and prevention techniques. The latter has been directly discussed in Section 7.7; in this section, we discuss detection techniques and other work that relates to ours.

XSS detection techniques focus on identifying holes in web application code that could result in vulnerabilities. Most of the vulnerability detection techniques have focused on server-side application code. We classify them based on the nature of the analysis, below.

- *Static and Quasi-static techniques.* Static analysis [54, 62, 76] and model checking techniques [75] aim to identify cases where the web application code fails to sanitize the input before output. Most static analysis tools are equipped with the policy that once data is passed through a custom sanity check, such as `htmpspecialchars` PHP function, then the input is safe. Balzarotti et al. [8] show that often XSS attacks are possible even if the developer performs certain sanitization on input data due to deficiencies in sanitization routines. They also describe a combined static and dynamic analysis to find such security bugs.

- *Server-side dynamic detection* techniques have been proposed to deal with the distributed nature of the server side checks. Taint-tracking [132, 16, 87, 90] on the server-side aims to centralize sanitization checks at the output interface with the use of taint metadata. These have relied on the assumption that server side processing is consistent with client side rendering, which is a significant design difference. These can be used as prevention techniques as well. Our work extends the foundation of taint-tracking to client-side tracking to eliminate difficulties of server-browser inconsistencies and to

safeguard client-side code as well. Some of the practical challenges that we share with previous work on taint-tracking are related to tracking taint correctly through multiple components of the web server platform efficiently. Cross-component taint tracking [85] and efficient designs of taint-tracking [98, 91, 70] for server-side mitigation are an active area of research which our architecture would readily benefit from.

Several other works have targeted fortification of web browser's same-origin policy enforcement mechanisms to isolate entities from different domains. Browser-side taint tracking is also used to fortify domain isolation [25], as well as tightening the sharing mechanisms such as iframe communication[13] and navigation. These address a class of XSS attacks that arise out of purely browser-side bugs or weak enforcement policies in isolating web content across different web page, whereas in this chapter, we have analyzed the class of reflected and stored XSS attacks only. MashupOS[120] discussed isolation and communication primitives for web applications to specify trust associated with external code available from untrusted source. Our work introduces primitives for isolation and confinement of inline untrusted data that is embedded in the web page.

Finally, the idea of parser-level isolation is a pervasively used mechanism. Prepared statements [36] in SQL are built on this principle, and Su et al. demonstrated a parser-level defense technique against SQL injection attacks[107]. As we show, for today's web applications the problem is significantly different than dealing with SQL, as untrusted data is processed dynamically both on the client browser and in the web server. The approach of using randomization techniques has been proposed for SQL injection attacks [19], control hijacking in binary code [64], and even in informal proposals for confinement in HTML using `<jail>` tag [20, 73]. Our work offers a comprehensive framework that improves on the security properties of `<jail>` element for static DSI (as explained in Section 7.3), and provides dynamic integrity as well.

7.10 Conclusion

We propose a defense approach that models XSS as a privilege escalation vulnerability, as opposed to a sanitization problem. It employs parser-level isolation for confinement of user-generated data through out the lifetime of the web application. We showed this scheme is practically possible in an architecture that is backwards compatible with current browsers. Our empirical evaluation over 5,328 real-world vulnerable web sites shows that our default policy thwarts over 98% of the attacks, and we explained how flexible server-side policies could be used in conjunction, to provide robust XSS defense with no false positives.

Chapter 8

Conclusion

In this thesis, we address the problem of automatically finding and preventing script injection vulnerabilities. Script injection vulnerabilities are a prominent class of web application flaws which are pervasive on today's web. These vulnerabilities affect both client- and server-side components of web applications. In this thesis, we make three contributions towards addressing this threat.

First, we propose two techniques for automatically finding these vulnerabilities in client-side JavaScript code. We develop techniques and systems to simplify dynamic analyses on JavaScript. The first technique is a single-path analysis of the application code, which combines dynamic taint tracking with fuzzing. It aims to find witness inputs that concretely demonstrate the presence of bug. In the second technique, we utilize dynamic symbolic execution with deeper reasoning of strings. This technique automatically generates a set of test cases that explores multiple paths in the application. In addition, it can reason about the application logic along any given path using a string decision procedure and find concrete exploit instances. We demonstrate that these techniques improve over prior work on testing for JavaScript flaws significantly and have found several real-world vulnerabilities. We hope that these techniques can further be utilized in testing other kinds of security flaws as well.

Second, we empirically study the use of sanitization, which is the predominant defense technique to prevent these attacks today. We uncover two new classes of errors in the practical use of sanitization in shipping web applications. We also find that emerging web application frameworks often do not provide any support for automatic sanitization. In some of web application frameworks that try to auto-sanitize, an unsafe strategy of context-insensitive sanitization is used.

As a third contribution, we propose a type-based approach to automatically perform correct sanitization for applications authored in emerging web application frameworks. We demonstrate the necessary security properties that an auto-sanitization must possess and build a proof-of-concept that can be used in an existing web templating framework. Finally, we also propose a sanitization-free defense for preventing script injection vulnerabilities as a second line of defense. The defense strategy aims to preserve a fundamental property of the application, which we call as document structure integrity. We discuss the feasibility of

deploying this mechanism with minimal impact to backwards compatibility under specific assumptions outlined in Chapter 7.

Bibliography

[1] Gisle Aas. "CPAN: URI::Escape". `http://search.cpan.org/~gaas/URI-1.56/URI/Escape.pm`.

[2] ab. "Apache HTTP server benchmarking tool". `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[3] *AdSafe : Making JavaScript Safe for Advertising*. `http://www.adsafe.org/`.

[4] alexa.com. "Alexa Top 500 Sites". `http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none`. 2008.

[5] Shay Artzi et al. "Finding bugs in dynamic web applications". In: *International Symposium on Software Testing and Analysis*. 2008.

[6] E. Athanasopoulos et al. "xJS: practical XSS prevention for web application development". In: *Proceedings of the 2010 USENIX conference on Web application development*. 2010.

[7] M. Balduzzi et al. "Automated discovery of parameter pollution vulnerabilities in web applications". In: *Proceedings of the 18th Network and Distributed System Security Symposium*. 2011.

[8] D. Balzarotti et al. "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications". In: *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA 2008.

[9] Sruthi Bandhakavi et al. *VEX: Vetting Browser Extensions For Security Vulnerabilities*. 2010.

[10] David Baron. *Mozilla's Quirks Mode*. URL: `https://developer.mozilla.org/en/mozilla's_quirks_mode`.

[11] Adam Barth, Juan Caballero, and Dawn Song. "Secure Content Sniffing for Web Browsers *or* How to Stop Papers from Reviewing Themselves". In: *Proceedings of the 30th IEEE Symposium on Security and Privacy*. Oakland, CA 2009.

[12] Adam Barth, Collin Jackson, and John C. Mitchell. "Robust Defenses for Cross-Site Request Forgery". In: *CCS*. 2008.

[13] Adam Barth, Collin Jackson, and John C. Mitchell. "Securing Frame Communication in Browsers". In: *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*. 2008.

[14] Adam Barth et al. *Protecting Browsers from Extension Vulnerabilities*. 2009.

[15] Daniel Bates, Adam Barth, and Collin Jackson. "Regular expressions considered harmful in client-side XSS filters". In: *Proceedings of the 19th international conference on World wide web*. WWW '10. 2010.

[16] Prithvi Bisht and V. N. Venkatakrishnan. "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2008.

[17] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. "Path Feasibility Analysis for String-Manipulating Programs". In: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2009.

[18] Hristo Bojinov, Elie Bursztein, and Dan Boneh. "XCS: Cross Channel Scripting and its Impact on Web Applications". In: *CCS*. 2009.

[19] Stephen W. Boyd and Angelos D. Keromytis. "Sqlrand: Preventing Sql Injection Attacks". In: *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*. 2004, pp. 292–302.

[20] CTO Mozilla Corp. Brendan Eich. *JavaScript: Mobility & Ubiquity*. Presentation. `http://kathrin.dagstuhl.de/files/Materials/07/07091/07091.EichBrendan.Slides.pdf`.

[21] J. Richard Büchi and Steven Senger. "Definability in the Existential Theory of Concatenation and Undecidable Extensions of this Theory". In: *Mathematical Logic Quarterly* 34.4 (1988), pp. 337–342.

[22] Juan Caballero et al. *Extracting Models of Security-Sensitive Operations using String-Enhanced White-Box Exploration on Binaries*. Tech. rep. UCB/EECS-2009-36. EECS Department, University of California, Berkeley, 2009.

[23] "CakePHP: Sanitize Class Info". `http://api.cakephp.org/class/sanitize`.

[24] Ashok Chandra et al. "Equations between regular terms and an application to process logic". In: *Proceedings of the 13th annual ACM Symposium on Theory of Computing (STOC)*. 1981, pp. 384–390.

[25] Shuo Chen, David Ross, and Yi-Min Wang. "An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism". In: *Proceedings of the 14th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2007, pp. 2–11. ISBN: 978-1-59593-703-2. DOI: `\url{http://doi.acm.org/10.1145/1315245.1315248}`.

[26] Ravi Chugh et al. "Staged information flow for JavaScript". In: *PLDI*. 2009.

[27] "ClearSilver: Template Filters". http://www.clearsilver.net/docs/man_filters.hdf.

[28] "CodeIgniter User Guide Version 1.7.2: Input Class". http://codeigniter.com/user_guide/libraries/input.html.

[29] "CodeIgniter/system/libraries/Security.php". https://bitbucket.org/ellislab/codeigniter/src/8af0fb079f90/system/libraries/Security.php.

[30] "Ctemplate: Guide to Using Auto Escape". http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html.

[31] CWE. "2011 CWE/SANS Top 25 Most Dangerous Software Errors". http://cwe.mitre.org/top25/. 2011.

[32] "django: Built-in template tags and filters". http://docs.djangoproject.com/en/dev/ref/templates/builtins.

[33] "Django Sites : Websites powered by Django". http://www.djangosites.org/.

[34] "ECMAScript Language Specification, 3rd Edition". http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[35] Matthew Finifter, Joel Weinberger, and Adam Barth. "Preventing Capability Leaks in Secure JavaScript Subsets". In: *Proc. of Network and Distributed System Security Symposium, 2010*. 2010.

[36] Harrison Fisk. "Prepared Statements". http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html. 2004.

[37] Fortify, Inc. *Fortify SCA*. http://www.fortifysoftware.com/products/sca/. 2006.

[38] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. "Flow-sensitive type qualifiers". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. PLDI '02. 2002.

[39] Vijay Ganesh and David L. Dill. "A Decision Procedure for Bit-Vectors and Arrays". In: *Computer Aided Verification, 19th International Conference (CAV)*. 2007, pp. 519–531.

[40] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *PLDI*. 2005.

[41] Patrice Godefroid, Michael Y. Levin, and David Molnar. "Automated Whitebox Fuzz Testing". In: *Network and Distributed System Security*. 2008.

[42] "Google AutoEscape Implementation for Ctemplate (C code)". http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html.

[43] "Google AutoEscape Implementation for GWT (Java code)". http://code.google.com/p/google-web-toolkit/source/browse/tools/lib/streamhtmlparser/streamhtmlparser-jsilver-r10/streamhtmlparser-jsilver-r10-1.5.jar.

[44] "Google Closure Templates". `http://code.google.com/closure/templates/`.

[45] Google Inc. "Issues: google-caja: A source-to-source translator for securing Javascript-based web content". `http://code.google.com/p/google-caja/issues/list?q=label:Security`.

[46] "Google Web Toolkit: Developer's Guide – SafeHtml". `http://code.google.com/webtoolkit/doc/latest/DevGuideSecuritySafeHtml.html`.

[47] Salvatore Guarnieri and Benjamin Livshits. "Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code". In: *Usenix Security*. 2009.

[48] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. "Using static analysis for Ajax intrusion detection". In: *Proceedings of the 18th international conference on World wide web*. WWW '09.

[49] Matthew Van Gundy and Hao Chen. "Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks". In: *16th Annual Network & Distributed System Security Symposium* (2009).

[50] Pieter Hooimeijer and Westley Weimer. "A decision procedure for subset constraints over regular languages". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2009, pp. 188–198.

[51] Pieter Hooimeijer et al. "Fast and Precise Sanitizer Analysis With BEK". In: *Proceedings of the Usenix Security Symposium*. 2011.

[52] "How To: Prevent Cross-Site Scripting in ASP.NET". `http://msdn.microsoft.com/en-us/library/ff649310.aspx`.

[53] "HTML Purifier : Standards-Compliant HTML Filtering". `http://htmlpurifier.org/`.

[54] Y Huang et al. "Securing Web Application Code by Static Analysis and Runtime Protection". In: *DSN* (2004).

[55] Yao-Wen Huang et al. "Securing web application code by static analysis and runtime protection". In: *Proceedings of the 13th international conference on World Wide Web*. WWW '04.

[56] IE 8 Blog: Security Vulnerability Research & Defense. "IE 8 XSS Filter Architecture and Implementation". `http://blogs.technet.com/swi/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx`. 2008.

[57] "iGoogle Gadget Directory". `http://www.google.com/ig/`.

[58] "Introducing JSON". `http://www.json.org/`.

[59] Susmit Jha, Sanjit A. Seshia, and Rhishikesh Limaye. "On the Computational Complexity of Satisfiability Solving for String Theories". In: *CoRR* abs/0903.2825 (2009).

[60] "JiftyManual". `http://jifty.org/view/JiftyManual`.

[61] T Jim, N Swamy, and M Hicks. "BEEP: Browser-enforced embedded policies". In: *16th International World World Web Conference* (2007).

[62] Nenad Jovanovic, Christopher Krügel, and Engin Kirda. "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)". In: *IEEE Symposium on Security and Privacy*. 2006.

[63] Kaluza. "Kaluza web page". `http://webblaze.cs.berkeley.edu/2010/kaluza/`. 2010.

[64] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization". In: *Proceedings of the 10th ACM conference on Computer and communications security*. 2003.

[65] Adam Kieżun et al. "Automatic creation of SQL injection and cross-site scripting attacks". In: *30th International Conference on Software Engineering (ICSE)*. 2009.

[66] Adam Kieżun et al. "HAMPI: A solver for string constraints". In: *International Symposium on Software Testing and Analysis*. 2009.

[67] Engin Kirda et al. "Noxes: a client-side solution for mitigating cross-site scripting attacks". In: *Proceedings of the ACM symposium on Applied computing*. 2006.

[68] Amit Klein. *DOM Based Cross Site Scripting or XSS of the Third Kind*. Tech. rep. Web Application Security Consortium, 2005.

[69] "kses - PHP HTML/XHTML filter". `http://sourceforge.net/projects/kses/`.

[70] Lap Chung Lam and Tzicker Chiueh. "A General Dynamic Information Flow Tracking Framework for Security Applications". In: *Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*. 2006.

[71] Benjamin Livshits and Monica S. Lam. "Finding Security Errors in Java Programs with Static Analysis". In: *Proceedings of the Usenix Security Symposium*. 2005.

[72] Benjamin Livshits, Michael Martin, and Monica S. Lam. *SecuriFly: Runtime Protection and Recovery from Web Application Vulnerabilities*. Tech. rep. Stanford University, 2006.

[73] Mike Ter Louw, Prithvi Bisht, and VN Venkatakrishnan. "Analysis of Hypertext Isolation Techniques for XSS Prevention". In: *Workshop on Web 2.0 Security and Privacy (W2SP)* (2008).

[74] Sergio Maffeis, John C. Mitchell, and Ankur Taly. "Object Capabilities and Isolation of Untrusted Web Applications". In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 125–140. ISBN: 978-0-7695-4035-1. DOI: `http://dx.doi.org/10.1109/SP.2010.16`. URL: `http://dx.doi.org/10.1109/SP.2010.16`.

[75] Michael Martin and Monica S. Lam. "Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking". In: *17th USENIX Security Symposium*. 2008.

[76] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. "Finding application errors and security flaws using PQL: a program query language". In: *Object-Oriented Programming, Systems, Languages, and Applications*. 2005.

[77] Armando B. Matos. "Periodic sets of integers". In: *Theoretical Computer Science* 127.2 (May 1994), pp. 287–312.

[78] Ali Mesbah, Engin Bozdag, and Arie van Deursen. "Crawling Ajax by Inferring User Interface State Changes". In: *Proceedings of the International Conference on Web Engineering*. 2008.

[79] Leo Meyerovich and Benjamin Livshits. "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser". In: *IEEE Symposium on Security and Privacy*. 2010.

[80] "Microsoft ASP.NET: Request Validation – Preventing Script Attacks". `http://www.asp.net/LEARN/whitepapers/request-validation`.

[81] Microsoft Corporation. *Microsoft Code Analysis Tool .NET*. `http://www.microsoft.com/downloads/en/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en`, 2009.

[82] Barton P. Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM*. 1990.

[83] Mozilla Foundation. "Tp2 Pageloader Framecycle Test". `http://mxr.mozilla.org/mozilla/source/tools/performance/pageload/`.

[84] Yacin Nadji, Prateek Saxena, and Dawn Song. "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense". In: *NDSS*. 2009.

[85] Susanta Nanda, Lap-Chung Lam, and Tzicker Chiueh. "Dynamic multi-process information flow tracking for web application security". In: *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*. 2007.

[86] James Newsome and Dawn Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software". In: *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*. 2005.

[87] A Nguyen-Tuong et al. "Automatically hardening web applications using precise tainting". In: *20th IFIP International Information Security Conference* (2005).

[88] NoScript. "NoScript". `http://noscript.net/`. 2008.

[89] OWASP. *OWASP Top 10 - 2010, The Ten Most Critical Web Application Security Risks*. Presentation. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[90] Tadeusz Pietraszek and Chris Vanden Berghe. "Defending Against Injection Attacks Through Context-Sensitive String Evaluation". In: *RAID*. 2004.

[91] Feng Qin et al. "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks". In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 2006.

[92] "Quasis Demo - JavaScript Shell 1.4". `http://js-quasis-libraries-and-repl.googlecode.com/svn/trunk/index.html`.

[93] W. Robertson and G. Vigna. "Static Enforcement of Web Application Integrity Through Strong Typing". In: *Proceedings of the USENIX Security Symposium*. Montreal, Canada 2009.

[94] RSnake. *XSS Cheat Sheet for filter evasion*. `http://ha.ckers.org/xss.html`.

[95] "Ruby on Rails Security Guide". `http://guides.rubyonrails.org/security.html`.

[96] Samy. "I'm Popular". Description of the MySpace worm by the author, including a technical explanation. 2005.

[97] Prateek Saxena, David Molnar, and Benjamin Livshits. "SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications". In: *Proceedings of the ACM Computer and communications security(CCS)*. 2011.

[98] Prateek Saxena, R Sekar, and Varun Puranik. "Efficient fine-grained binary instrumentationwith applications to taint-tracking". In: *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*. 2008.

[99] Prateek Saxena et al. *A Symbolic Execution Framework for JavaScript*. Tech. rep. UCB/EECS-2010-26. EECS Department, University of California, Berkeley, 2010.

[100] Prateek Saxena et al. "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications". In: *17th Annual Network & Distributed System Security Symposium, (NDSS)*. 2010.

[101] SecuriTeam. "Google.com UTF-7 XSS Vulnerabilities". `http://www.securiteam.com/securitynews/6Z00L0AEUE.html`. 2008.

[102] R. Sekar. "An Efficient Black-box Technique for Defeating Web Application Attacks". In: *NDSS*. 2009.

[103] "Smarty Template Engine: escape". `http://www.smarty.net/manual/en/language.modifier.escape.php`.

[104] Dawn Song. "RISE: Randomization Techniques for Software Security". In: *Presentation at Software Security Summer Institute, http://www.cs.berkeley.edu/~dawnsong/papers/rise.pdf*. June 2003.

[105] Brandon Sterne and Adam Barth. *Content Security Policy: W3C Editor's Draft*. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`. 2012.

[106] Elizabeth Stinson and John C. Mitchell. "Characterizing Bots' Remote Control Behavior". In: *Botnet Detection*. 2008.

[107] Zhendong Su and Gary Wassermann. "The essence of command injection attacks in web applications". In: 2006.

[108] Symantec Corp. *Symantec Internet Security Threat Report*. Tech. rep. Symantec Corp., 2008. URL: `\url{http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_exec_summary_internet_security_threat_report_xiii_04-2008.en-us.pdf}`.

[109] "Template::Manual::Filters". `http://template-toolkit.org/docs/manual/Filters.html`.

[110] Ter Louw, Mike and V.N. Venkatakrishnan. "BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2009.

[111] "The Django Book: Security". `http://www.djangobook.com/en/2.0/chapter20/`.

[112] "The Mason Book: Escaping Substitutions". `http://www.masonbook.com/book/chapter-2.mhtml`.

[113] TwitPwn. "DOM Based XSS in Twitterfall". In: (2009). URL: `http://www.twitpwn.com/2009/07/motb-08-dom-based-xss-in-twitterfall.htm`.

[114] Unicode, Inc. "Unicode Character Database". `http://unicode.org/Public/UNIDATA/PropList.txt`. 2008.

[115] "UTF-7 XSS Cheat Sheet". `http://openmya.hacker.jp/hasegawa/security/utf7cs.html`.

[116] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. "Rex: Symbolic Regular Expression Explorer". In: *International Conference on Software Testing, Verification and Validation*. 2010.

[117] Wietse Venema. "Taint support for PHP". `ftp://ftp.porcupine.org/pub/php/php-5.2.3-taint-20071103.README.html`. 2007.

[118] P. Vogt et al. "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In: *Proceeding of the Network and Distributed System Security Symposium (NDSS)*. San Diego, CA 2007.

[119] W3C. "HTML5 Specification". `http://www.w3.org/TR/html5/`.

[120] Helen J. Wang et al. "Protection and communication abstractions for web browsers in MashupOS". In: *SOSP*. 2007.

[121] Gary Wassermann et al. "Dynamic test input generation for web applications". In: *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*. 2008.

[122] Gary Wassermann et al. "Dynamic test input generation for web applications". In: *Proceedings of the International symposium on Software testing and analysis*. 2008.

[123] Web Application Security Consortium. "Web Application Security Statistics Project 2007". `http://www.webappsec.org/projects/statistics/wasc_wass_2007.pdf`.

[124] Joel Weinberger, Adam Barth, and Dawn Song. *Towards Client-side HTML Security Policies*.

[125] Joel Weinberger et al. "A Systematic Analysis of XSS Sanitization in Web Application Frameworks". In: *Proceedings of the European Symposium on Research in Computer Security*. 2011.

[126] Joel Weinberger et al. *An Empirical Analysis of XSS Sanitization in Web Application Frameworks*. Tech. rep. UCB/EECS-2011-11. EECS Department, University of California, Berkeley, 2011.

[127] Yichen Xie and Alex Aiken. "Static Detection of Security Vulnerabilities in Scripting Languages". In: *Proceedings of the Usenix Security Symposium*. 2006.

[128] *XML Path Language 2.0*. `http://www.w3.org/TR/xpath20/`.

[129] "XSS Prevention Cheat Sheet". `http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet`.

[130] XSSed.com. "Famous XSS Exploits". `http://xssed.com/archive/special=1`. 2008.

[131] "xssterminate". `http://code.google.com/p/xssterminate/`.

[132] Wei Xu, Sandeep Bhatkar, and R Sekar. "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks". In: *USENIX Security Symposium* (2006).

[133] "Yii Framework: Security". `http://www.yiiframework.com/doc/guide/1.1/en/topics.security`.

[134] Dachuan Yu et al. "JavaScript instrumentation for browser security". In: *SIGPLAN Not.* 42.1 (2007), pp. 237–249. ISSN: 0362-1340. DOI: `http://doi.acm.org/10.1145/1190215.1190252`.

[135] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. "Symbolic String Verification: Combining String Analysis and Size Analysis". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2009, pp. 322–336.

[136] M. Zalewski. "Browser security handbook". In: *Google Code* (2010). `http://code.google.com/p/browsersec/wiki/Part1`.

[137] "Zend Framework: Zend_Filter". `http://framework.zend.com/manual/en/zend.filter.set.html`.