UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Analysis and Enforcement of Web Application Security Policies

Permalink
https://escholarship.org/uc/item/50v7m8wc

Author
Weinberger, Joel Howard Willis

Publication Date
2012

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Analysis and Enforcement of Web Application Security Policies

by

Joel Howard Willis Weinberger

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 Brian Carver
Professor David Wagner

Fall 2012

Analysis and Enforcement of Web Application Security Policies

Copyright 2012
by
Joel Howard Willis Weinberger

Abstract

Analysis and Enforcement of Web Application Security Policies

by

Joel Howard Willis Weinberger

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

Web applications are generally more exposed to untrusted user content than traditional applications. Thus, web applications face a variety of new and unique threats, especially that of content injection. One method for preventing these types of attacks is web application security policies. These policies specify the behavior or structure of the web application. The goal of this work is twofold. First, we aim to understand how security policies and their systems are currently applied to web applications. Second, we aim to advance the mechanisms used to apply policies to web applications. We focus on the first part through two studies, examining two classes of current web application security policies. We focus on the second part by studying and working towards two new ways of applying policies. These areas will advance the state of the art in understanding and building web application security policies and provide a foundation for future work in securing web applications.

To my wife, Sarah.

Without you, I would be nowhere.

Contents

Contents ii

List of Figures iv

List of Tables v

1 Introduction 1
- 1.1 Web Application Security Policies 2
- 1.2 Current Policy Mechanisms 4
- 1.3 Policy Application 8

2 Background and Related Work 12

3 Evaluating Implicit Policy Mechanisms in Web Frameworks 15
- 3.1 Introduction 15
- 3.2 A Systematic Browser Model for XSS 17
- 3.3 Analysis of Web Frameworks and Applications 22
- 3.4 Related Work 31
- 3.5 Conclusions and Future Work 32

4 Evaluating the Effectiveness of Content Security Policy 35
- 4.1 Introduction 35
- 4.2 HTML Security Policies 36
- 4.3 Evaluating the Application of CSP 39
- 4.4 Related Work 42
- 4.5 Towards HTML Security Policies 43
- 4.6 Conclusion 44

5 Static Enforcement of Policies for Advertisements 45
- 5.1 Introduction 45
- 5.2 Related Work 48
- 5.3 Statically Verified Containment 51
- 5.4 Detecting Containment Breaches 53

5.5 Blancura 59
5.6 Conclusions 62

6 Translation of JavaScript Towards Verification 64
6.1 Introduction 64
6.2 A brief review of F* 67
6.3 A library for dynamic typing in F* 69
6.4 Translating JavaScript to ML 73
6.5 Examples 76
6.6 Using the Translation for Verifying Correctness and Security 79
6.7 Related Work 80
6.8 Conclusions 81

7 Conclusion 83

A Transductions in the Browser 85

B Alexa US Top 100 87

Bibliography 90

List of Figures

1.1 A publisher sells space on his or her page to an advertising network. This space may be resold through multiple advertising networks, until it is sold to an advertiser, who provides an advertisement written in a secure JavaScript subset. The advertisement is checked for safety by each advertising network, in turn, and ultimately served to visitors of the publisher's web page. 9

3.1 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. 18

5.1 A publisher sells space on his or her page to an advertising network. This space may be resold through multiple advertising networks, until it is sold to an advertiser, who provides an advertisement written in a secure JavaScript subset. The advertisement is checked for safety by each advertising network, in turn, and ultimately served to visitors of the publisher's web page. 46
5.2 Depiction of a simple heap graph. In (a), all nodes are vetted nodes. In (b), a suspicious edge has been detected. The host code has added a pointer that provides the guest code with access to an unvetted object. The suspicious edge points from the prototype of a built-in object to a method defined by the host. . 54
5.3 Visual depictions of the results of the experiment. 57
5.4 Exploit for `people.com` 58
5.5 Exploit for `twitter.com` 60

6.1 A JavaScript program (top) and its MLjs version 66
6.2 A refinement of type dynamic 70
6.3 Some basic JavaScript primitives in JSPrims 71
6.4 Some DOM primitives in JSPrims 72
6.5 A portion of the JS2ML translation. Note that the pos type represents a program position in the original JavaScript program. Also, for brevity, we elide the definition of value which defines the grammar for JavaScript primitives. 73

List of Tables

3.1 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. 24
3.2 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. 25
3.3 Sanitizers provided by languages and/or frameworks. For frameworks, we also include sanitizers provided by standard packages or modules for the language. . 28
3.4 The web applications we study and the contexts for which they sanitize. 29

4.1 Percent difference in performance between modified Bugzilla and original with 95% confidence intervals. 41
4.2 Percent difference in performance between modified HotCRP and original with 95% confidence intervals and jQuery Templating performance. 41

5.1 Slowdown on the "read" and "write" micro-benchmarks, average of 10 runs. . . 51

A.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 3.1. 86
A.2 Details regarding the transducers mentioned in Table A.1. They all involve various parsers and serializers present in the browser for HTML and its related sub-grammars. 86

B.1 Subset of the Alexa US Top 100 sites used in the experiment. 89

Acknowledgments

My first thanks go to my adviser, Dawn Song, for her great advice and help in working towards this thesis. Also, thanks to the rest of my thesis committee, Brian Carver and David Wagner, as well as Koushik Sen from my qualification exam committee, for their great help in directing this work.

A special thanks to all of my coauthors who contributed to the chapters of this thesis: Devdatta Akhawe, Adam Barth, Juan Chen, Matthew Finifter, Ben Livshits, Prateek Saxena, Cole Schlesinger, Richard Shin, Dawn Song, and Nikhil Swamy. I have been blessed to be surrounded by and work with such brilliant people over the past five years.

After all these years of academic work, I have had the pleasure of taking courses from many wonderful teachers and professors. However, I want to particularly thank three teachers who, over the years, profoundly affected the way I work and the way I think. I cannot thank them enough for what they contributed to my education. Mr. Jon Campbell, of Columbia High School, my high school history and government teacher, showed that you could never ask enough questions or dig deep enough into a problem. Professor Shriram Krishnamurthi, of Brown University, has an unmatched passion for computer science education, which comes across every day in his course, and he brilliantly demonstrated the power of the intersection of practical and theoretical thought in his programming languages course. Professor Brian Carver, of the University of California, Berkeley, opened a new world of thought about law to me in his wonderful intellectual property law course; in another life, I would have been a lawyer.

Most importantly, thanks must be given to my family, who have tirelessly supported me through the years.

My wife's parents' and siblings are the most caring family I could have dreamed of marrying into. They are an incredible group of people, who I cannot imagine living without.

My parents, Meredith Sue Willis and Andrew Weinberger, could not have been a more supportive and loving duo, and all those years of arguing over the dinner table are finally paying off. Their tireless energy, from cross-country meets to tap dance recitals to listening to my rants and complaints across the country, has meant more to me than they will ever know.

Finally, my incredible wife, to whom all my work, past and future, is dedicated, Sarah Weinberger. Everything I accomplish is because of your love and support.

Chapter 1

Introduction

As the web becomes increasingly complex, web applications become more sophisticated and dynamic. One of the most important ways that web applications have become complex is in how they use input data. Web pages are no longer static; they contain dynamic content from sources that may be trusted, untrusted, or trusted but potentially buggy. There are many places that this type of data can come from: user input, advertisements, or widgets, to name a few. These sources of data have led to a class of attacks know as *content injection attacks*. In these attacks, an attacker is able to place malicious content on a page and make it act as if it came from the developer. This can lead to *cross-site scripting* attacks, *phishing*, or malicious information.

In order to counter these types of attacks, developers implement *web application security policies* — policies defining allowed and disallowed behavior of content on a web page. Traditionally, these policies have been implicitly defined by the use of *sanitization functions*, or *content filters*, that modify untrusted data to conform to a well-understood and safe set of behaviors. However, this approach is notoriously brittle and sometimes impossible, depending on what type of behavior is enforced. At the very least, this approach requires deep understanding of application semantics and the browser model by developers.

Several alternative solutions have been proposed for different types of content. Increasingly, web frameworks are used to automate the application of sanitizers to HTML content. For more dynamic and context specific content, such as advertisements, verifiers and dynamic enforcement mechanisms like Caja [31] and ADsafe [41] have been proposed. Alternatively, there have been a number of proposals, such as BLUEPRINT [120] and Content Security Policies [113], to make these HTML security policies explicit in web applications.

However, none of these approaches addresses the need to analyze what the security policies of web applications actually are. That is to say, all of these approaches apply mechanisms without asking what the policy *is*. Furthermore, many of these solutions have a variety of problems that prevent developers from fully embracing them.

We propose that tools can be built to greatly improve the analysis and understanding of web application security policies, as well as be used to better enforce such policies, both on legacy and new applications. We can build tools to help developers understand the policies,

implicit and explicit, that they write in applications.

1.1 Web Application Security Policies

Web applications currently use a variety of mechanisms to enforce web application security policies, albeit usually implicitly. For the most part, enforcement of these policies falls to the hands of sanitization, the basic filtering of untrusted content. Research has proposed several new tools for enforcing policies, but with limited exceptions, these have not been adopted by the development community[1].

In modern HTML based web applications, much of the untrusted content is placed in the HTML context. Enforcement of policies often falls to either the manual application of sanitizers, or to a web framework that attempts to aid the developer in filtering the untrusted content. However, little has been done to measure the effectiveness of these tools or even what the real requirements of web applications are. As a first step, we study these questions and show that the types of policies that web applications try to enforce cannot be handled by current web application frameworks.

Even if these frameworks had more powerful policy enforcement system for HTML, it would not help with the growth in untrusted script content that is placed on pages. For example, advertisements with dynamic JavaScript content are placed by a variety of advertisers on web pages. The advertisement networks distributing these ads require ways to constrain the ads so they cannot interact maliciously with the hosting page while still providing dynamic content. In response, there have been several solutions developed, both static and dynamic, such as Yahoo!'s ADsafe and Google's Caja platform. However, these systems lack several important properties, including the ability to enforce dynamic policies. We examine the current state of these enforcement mechanisms and what they lack.

Of particular note is that none of this systems, for HTML or script policies, help the programmer to understand what the policy of legacy applications actually *is*. BLUEPRINT and CSP make policies explicit, and CSP also attempts to apply policies to legacy applications. Thus, our initial focus is on understanding current web application security policies.

In order to improve web application security policies, we focus on two primary areas: the extraction of policies from web applications and the application of policies to web applications. This aids both the understanding and improvement of current and legacy applications as well as the development of new applications.

Current Policy Systems Several systems are available that attempt implement web application security policies, both implicitly and explicitly. Our first step is to examine these systems that currently implement web application security policies and to measure their power and effectiveness. This will help us better understand the problem space, as well as the advantages and pitfalls of current solutions.

[1]Content Security Policy is the notable exception here as it is currently implemented and released in all of the major browsers. We discuss Content Security Policy in more detail in Chapter 4.

We focus on providing the first evaluations of two distinct but very important areas of web application security policies. The first approach are web frameworks that generally implement and encourage the use of *sanitizer functions*. These functions remove untrusted elements from untrusted data they are provided with. This implicitly defines a web application security policy because the syntactic removal defines a set of banned behaviors at that point in the application. In Chapter 3, we evaluate the effectiveness of web frameworks in sanitizing untrusted data in web applications, showing that the power they offer is well short of what is required by web applications. We propose several ways in which they can improve the mechanisms presented to the developer.

The second, more recent approach is the application of explicit HTML policies, provided by systems such as BEEP [76], BLUEPRINT [120], and Content Security Policies (CSP) [113]. These systems provide explicit mechanisms for stating how parts of applications should behave in order to contain untrusted data. In Chapter 4, we show that the systems currently in place are powerful, but occupy two extreme ends of a spectrum, namely that of BLUEPRINT, an ultra-fine grained approach at the expense of performance, and CSP, a very coarse grained approach at the cost of usability.

Policy Application While many new languages, frameworks, and tools provide mechanisms to enforce behavioral limitations on web applications, little has been done to analyze the implicit policies of current applications that do not use such policy tools. For example, the PHP language [101] does not natively support web application policies explicitly so web developers generally need to resort to the manual application of sanitizers. While some frameworks aim to help by adding auto-sanitization and other features [141, 144], many applications have been and still are written outside of these frameworks.

This also applies to higher level systems. For example, safe JavaScript subsets, such as ADsafe, make claims about application behavior and how they constrain code that is written in the subset. Little has been done to measure the accuracy of these claims. Another way of looking at this problem is, does the stated policy match the actual, enforced policy?

This is a form of policy application. Many current web applications are written without explicit web application security policies because they are written in languages that do not support policies, or without tools for analyzing policies. For example, PHP contains support for policies only through coarse-grained, manual mechanisms, such as `htmlspecialchars`. These legacy applications are already built with manual policies applied, which is why we propose policy extraction to aid our understanding of them.

However, for new applications, we can build novel techniques and tools for writing web application security policies. One way to do this is to provide domain specific languages and verifiers to enforce policy. While many proposals have focused on more general solutions, we focus on more domain specific solutions in the belief that this, ultimately, allows the developer to write more flexible and powerful policies.

In this case, we focus on enforcing policies for JavaScript, but from two different perspectives. First, we look at a current JavaScript static enforcement mechanism, ADsafe, and try

to determine if the policies that are applied match the actual stated goals of the enforcement mechanism. In the second, we work on a novel technique for translating JavaScript into a new language that can use static type checking to enforce policies.

In Chapter 5, we focus on "safe JavaScript subsets," such as ADsafe. We look at ADsafe, in particular, which attempts to provide a security policy that can be statically enforced. We systematically explore this subset to understand if the enforced policy matches the proposed policy. We then propose a new static policy mechanism based on ADsafe to enforce policies.

After this, in Chapter 6, we look at a new technique towards enforcing policies on small JavaScript programs. We develop a new for translating JavaScript into a separate language for policy enforcement. In this language, type checking can be used to statically enforce policies. We particularly focus on small JavaScript programs, such as browser extensions.

1.2 Current Policy Mechanisms

Researchers have been looking at web application security policies for some time now, and several mechanisms for applying policies have been proposed and implemented. In this section, we examine this design space in order to better identify the problems at hand and the value of current solutions.

As web applications become more complex, web frameworks provide a structured way to ease development. These frameworks, such as Django [48] and Ruby on Rails [104], offer support for web application security policies in the form of sanitization. However, it is unclear if the support they provide is sufficient for the security policy needs of applications. We propose to study this question, comparing the needs of web applications to the features that web applications provide. In addition, we propose to develop the first formal model of the internals of a web browser as it pertains to XSS vulnerabilities.

Aside from web frameworks, a great deal of research has been aimed at another method of applying web application security policies. These mechanisms apply *explicit* policies, as opposed to the implicit policies based on sanitization of web frameworks. Examples of these systems are BEEP [76], BLUEPRINT [120], and Content Security Policies (CSP) [113]. In this work, we study the space that these systems fall in to understand the affects of applying policies in these systems. In particular, we perform the first detailed study of CSP to assess its strengths and weaknesses, and to find a new point in the design space of explicit web application security policies.

Web Frameworks

Web frameworks are systems, often implemented as libraries, built around a programming language to provide a structured way to implement web applications. Many web frameworks now provide mechanisms to support web application security policies, particularly in the HTML portion of applications. Researchers have proposed many defenses, ranging from purely server-side to browser-based or both [28, 98, 103, 85, 80, 129, 18]. However,

sanitization or *filtering*, the practice of encoding or eliminating dangerous constructs in untrusted data remains the industry-standard defense strategy in existing applications [100]. Web frameworks can offer a platform to make the sanitization in web applications, which was initially fairly ad hoc and manual, more systematic.

As part of this work, we clarify the assumptions and basic primitives web frameworks use today and explain the challenges and subtleties in XSS sanitization that web developers currently face. To define XSS systematically, we present the first systematic model of the web browser internals as it pertains to XSS vulnerabilities. An XSS attack occurs when the web browser parses data controlled by a low-privileged entity as high-privileged application code. Our browser model includes details of the various sub-languages of the web platform, their internal interactions, and the modifications browsers make dynamically. These intricacies impact the assumptions made by sanitization routines in web applications. We present, in comprehensive detail, why XSS sanitization can be subtle. For instance, we show that sanitization performed on the server-side may be "undone" during the browser's parsing of content into the DOM, which affects the security of client-side JavaScript code. Sanitizing correctly involves understanding how the web browser parses and interprets web content — an important issue not fully explained in prior XSS security research.

A web framework can, in principle, address XSS using sanitization if it addresses the subtleties we outline in our browser model. Whether existing frameworks achieve this goal today is the subject of this study. We compare the techniques and state of modern web frameworks to the empirically-observed requirements of large web applications. We systematically study the XSS sanitization both in web frameworks and existing application along the following dimensions:

- **Context Expressiveness and Sanitizer Correctness**. Untrusted data can be embedded in web application output in many different *contexts*. In fact, sanitizer which provide safety for one context do not protect data embedded in other contexts. For example, untrusted data appearing in an HTML tag content must be sanitized differently from input that appears in a URI attribute of an HTML tag. *Do web frameworks provide correct sanitizers for different contexts that applications commonly use in practice?*

- **Context-Sensitive Auto-sanitization Support**. Applying sanitizers in code automatically, which we term *auto-sanitization*, shifts the burden of ensuring safety against XSS from developers to frameworks. *Do web frameworks offer auto-sanitization at all?*. Further, a sanitizer that may be safe for use in one context may be unsafe for use in another. Therefore, to achieve security, sanitization must be *context-sensitive*; otherwise, auto-sanitization may provide a false sense of security. *Do web frameworks adequately address the requirement of context-sensitivity while auto-sanitizing sanitization?*

- **Mediation in client-side code**. AJAX applications have significant client-side code components, such as in JavaScript. There are numerous subtleties in XSS sanitiza-

tion because client-side code may read values from the DOM. *Do frameworks support complete mediation on DOM accesses in client-side code?*

We perform an empirical analysis over 14 mature web application development frameworks, most of which are used heavily in commercial applications, and 8 large, popular open-source web applications to analyze security along the dimensions outlined above. We provide the first in-depth study of the gap between the sanitization provided by web frameworks and what web applications require to correctly sanitize inputs.

We explain the subtleties of XSS in a comprehensive way. We study modern web frameworks' support for sanitizing user input and show that frameworks vary widely in the types of sanitization they support. By examining the use of sanitizers in a set of real-world applications, we identify a gap between the sanitizers provided by frameworks and those required by web applications.

Our results show that, unfortunately, the current set of web frameworks are inadequate for expressing web application security policies; they do not provide sufficient expressiveness for the set of policies that web applications actually look to enforce. However, we are also able to determine a minimum set of requirements that we believe can serve as a reasonable starting point for web application frameworks to express.

Explicit Policy Systems

Cross-site scripting (XSS) attacks are a significant threat to web applications today [136, 5]. XSS is part of a larger category of content injection attacks that threaten web applications. These attacks violate the integrity of web applications, steal from users and companies, and erode privacy on the web. Researchers have focused a great deal of effort at preventing these attacks, ranging from static detection to dynamic checking of content. One recent set of proposals that addresses content injection, *HTML security policies*, has become popular [120, 113]. HTML security policies provide a mechanism for an application to specify how a document is parsed and interpreted by the browser and can be viewed as a subproblem of more general web application security policies as they relate to HTML code. Specifically, HTML security policies are mechanisms that define how untrusted data is handled by a web application, such that untrusted data cannot change the structure of the page in unintended ways. Thus, they are meant as a defense against cross-site scripting and other types of content injection attacks by controlling how untrusted data can affect the behavior and structure of a web page.

HTML security policies are attractive for several reasons. They can help mitigate bugs on the server, either in how structure of content is determined or in the programmer's model of what content can and should be. There always might be bugs in a security policy as well, but the hope is that policies will be simpler, more explicit, and easier to understand. Additionally, while servers usually perform sanitization on content, it is not always obvious what sanitization method to apply in what context [107]. HTML security policies allow the programmer to state what *structure* should be allowed in a document.

There are are several proposals for HTML security policy systems. These systems make trade-offs in granularity to achieve different goals. However, these trade-offs are not explicitly discussed in the literature and are not well understood. The systems have different levels of how and where policies are applied, whether on the entire page or individual elements on the page. We show that this granularity can affect the performance of the policy system and how difficult it is to apply policies to a real application.

We analyze this space and to find a new point in the trade-offs of HTML security policy systems. We evaluate a state-of-the-art policy system and how its trade-offs affect real world use of of the system, and to understand the strengths and weaknesses of this system.

Several HTML security policy systems have been proposed, notably BLUEPRINT [120] and Content Security Policy (CSP) [113]. We examine these systems to understand their respective power and limitations. While they both provide working systems for expressing and implementing HTML security policy, CSP is of particular importance because it is deployed in the Firefox [12], Safari [13], and Chrome [11] web browsers.

Evaluation of the Application of CSP We evaluate the efficacy of CSP, both as an HTML security policy foundation and the practicalities of implementing applications with it. We discuss what CSP offers as a policy system, and the benefits of this approach. We also show several problems with CSP:

- We show that the application of CSP to prevent XSS in legacy applications is not necessarily a simple task. We show that in the applications we examine, they require a large number of modifications to make them work with a CSP policy that prevents XSS, in particular because of the requirement to remove all inline scripts.
- The use of CSP to prevent XSS attacks on an application requires modifications to an application that may cause large performance hits. In our evaluation, we show performance costs of up to 50% slowdown compared to the unmodified performance.

We demonstrate this by retrofitting two real applications, Bugzilla [1] and HotCRP [2], to use CSP. We evaluate the cost of our modifications with a number of metrics, including code change and performance metrics. We also discuss the process of modifying the applications, identifying several of the more common problems encountered.

Using this analysis of the strengths and weaknesses, we discuss the trade offs in HTML security policy systems and identify a new set of properties for a new system to implement. This analysis has been completed and is being prepped for submission. We show in our evaluation that the performance of our modified versions of the example applications experience substantial slowdowns in performance.

1.3 Policy Application

Web application security policies protect applications from untrusted user content by specifying the allowed behaviors of the content. In current web applications, these policies are generally implicitly defined by the what the application does to this content— namely, how the application applies sanitizers. Unfortunately, this means that the policies are often not crisply defined, and thus not well understood by developers or designers.

While many languages provide at least basic support for sanitization, such as PHP's `htmlspecialchars` function, there has been little work on understanding what the policies are. The developer may use a web application framework, but ultimately, the policy is still implicitly defined by their own labeling and annotations. Thus, the developer is required to do their own manual audit to ensure a safe web application security policy.

This is a problem for several reasons. First, a developer may intend to write a policy with several different code paths, but the same set of sanitizers applied to data. However, it is not uncommon for mistakes to be made, and for different sanitizers to be applied on different paths unintentionally. Second, a developer may code a set of policies for the untrusted data but not explicitly write them down. At a later point, she may not remember the policies that apply to certain data or another developer may need to know this. Because the policies are not known, developers are forced to manually understand the policy.

Related to this problem are policies for more dynamic content, such as JavaScript. Interestingly, web application security policy verifiers for scripts are more prevalent than for HTML content. However, while the policies of these verifiers are generally well defined, their correctness has not been checked. Thus, there is little reason to trust the policy of the verifiers or the implementation.

Additionally, applications already written in today's frameworks and with today's tools are limited in how they can express policy. We can provide useful tools for improving our understanding of how these applications use web application security policies, but this does not necessarily improve the mechanisms by which these applications can express policies. For new applications, there is more flexibility in the types of tools we can provide to help developers express web application security policies.

Towards addressing these problems, we propose two new techniques. The first extracts the enforced policy of the static script verifier, ADsafe, to understand the properties it actually provides. We discovered that the policy it enforces falls short of what developers might expect or desire. We then develop a new version of ADsafe with the desired properties. To aid in this, we also develop a tool that tracks the heap structure of the web browser to detect connections between different security principles.

Our second technique is a novel translation of JavaScript to ML. We show that this technique is a promising future direction for fine-grained policy enforcement for JavaScript. We show through several examples the power of this translation, and that it is promising for using type checking to prove behaviors of programs.

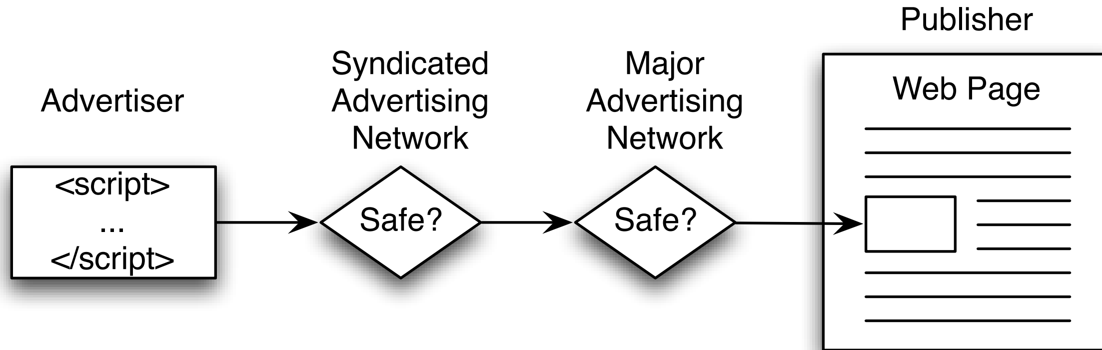

Figure 1.1: A publisher sells space on his or her page to an advertising network. This space may be resold through multiple advertising networks, until it is sold to an advertiser, who provides an advertisement written in a secure JavaScript subset. The advertisement is checked for safety by each advertising network, in turn, and ultimately served to visitors of the publisher's web page.

Verifying Verifier Policies

At its core, displaying third-party advertisements on a publisher's web site is a special case of a mashup. In this work, we study whether existing techniques are well-suited for containing advertisements and propose improvements to mashup techniques based on static verifiers. We study the existing techniques by applying dynamic analysis of the application's heap structure to verify if the *policy* of the static verifier matches the *goals* of the verifier; that is, do the verifiers correctly enforce the intended policy?

Static verifiers, such as ADsafe [41], Dojo Secure [145], and Jacaranda [44], are a particularly promising mashup technique for advertising. In this approach, the advertising network (and one or more of its syndicates) verifies that the advertisement's JavaScript source code conforms to a particular subset of the JavaScript language with desirable containment properties (see Figure 5.1). The precise JavaScript subset varies between systems, but the central idea is to restrict the guest advertisement to a well-behaved subset of the language in which the guest can interact only with object references explicitly and intentionally provided to the guest by the host, thus preventing the guest from interfering with the rest of the page.

We focus on evaluating and improving the containment of safe JavaScript subsets that use static verification. Static verifiers are appealing because they provide fine-grained control of the advertisement's privileges. For example, the hosting page can restrict the advertisement to instantiating only fully patched versions of Flash Player, preventing a malicious advertisement from exploiting known vulnerabilities in older versions of the plug-in. However, existing static verifiers do not provide perfect containment. To properly contain advertisements, these systems impose restrictions on the *publisher*: the publisher must avoid using certain JavaScript features that let the advertisement breach containment.

These static verifiers impose restrictions on publishers because of a design decision shared by the existing static verifiers: the verifiers restrict access to object properties using a static blacklist. The blacklist prevents the guest from accessing properties, such as `__proto__`, that can be used to breach containment. The designers of the subsets warrant that their blacklist is sufficient to prevent an advertisement from breaching containment on an otherwise empty page in their supported browsers (or else the subset would always fail to contain advertisements), but the blacklist approach does not restrict access to new properties introduced by the publisher. For example, a publisher might add a method to the string prototype that has not been vetted by the subset designer.

This blacklist defines the HTML application security policy that the verifier wishes to enforce. The verifier guarantees that applications it passes will not be able to violate these bounds; that is, those applications will not be able to access properties on the blacklist. In this study, we assume the verifier is implemented correctly. Our questions are only regarding the effectiveness of this policy.

The policy that ADsafe enforces makes several assumptions about the behavior of host websites. Namely, if the host website modifies the prototypes of any common objects, the ADsafe's policy fails to be secure. ADsafe has, in the past, stated that it assumes most web pages do not code in such a way. To measure if these assumptions hold, we build a tool that dynamically analyzes the heap graph of web pages running advertisements verified by ADsafe. The tool tracks the connections between different objects in the JavaScript heap, identifying them as part of the advertisement or the host page. If the ADsafe assumptions fail to hold, then advertisements will be able to break out of their contained space through unrestricted connections in the heap graph.

We run our tool on the non-adult web sites in the Alexa US Top 100 [15] to determine (1) how often sites add methods to prototypes and (2) how many of these methods could be used to breach containment. We discover that, of the non-adult web sites in the Alexa US Top 100, 59% expose new properties to the guest and that 37% of these sites contained at least one method that could be exploited by an ADsafe-verified advertisement to mount a cross-site scripting (XSS) attack against the publisher.

While publishers could manually vet their exposed methods, we propose eliminating this attack surface by replacing the property blacklist with a whitelist, making static verifiers more robust. Using a technique similar to script accenting [33], we prevent the guest advertisement from accessing properties defined by the hosting page unless those properties are explicitly granted to the guest. We show that our safe JavaScript subset is as expressive and as easy-to-use as ADsafe by implementing a simple, idempotent compiler that transforms any ADsafe-verified guest into our proposed subset.

Thus, in this work, we check the actual policy of a set of static JavaScript verifiers. We show that the policy of the verifiers is, in fact, insecure. We then propose an alternate web application security policy that we show is secure in the same scenarios.

Translation for Policies via Type Checking

While policies can be used to prevent untrusted code from executing in an application, there are times when developers want untrusted code to execute on the page, albeit in a controlled way. Advertisements are one such example, but also consider web browser extensions. In these examples, an untrusted third party provides a JavaScript snippet for the browser to execute, and as such, it may want to guarantee certain limitations on what that code can do.

Several systems have been developed to enforce web application policies on these untrusted JavaScript scripts. Most notably are Yahoo!'s ADsafe [41] and Google's Caja [31]. ADsafe, described in Section 1.3, provides a static verifier to ensure that the script conforms to a safe JavaScript subset. On the other hand, Caja provides a dynamic verifier that compiles the script and checks its behavior as it executes. Both approaches have limitations. In the case of ADsafe, the verifier is based on syntactic checks with limited insight into the application's behavior. Thus, the application is potentially limited to a stricter web application security policy than may be necessary. On the other hand, Caja provides a much richer set of semantics at the cost of performance. While ADsafe's verifier is purely static, Caja compiles in a number of dynamic checks that add substantially to the overhead of the application [53].

In this work, we develop a novel translation of JavaScript to ML. We show that this technique is promising for enforcing fine-grained policies on JavaScript applications. Through several examples of JavaScript browser extensions, we show that type checking techniques on translated JavaScript applications have great potential.

Recently, Guha et. al. developed a system for verifying web application security policies for web browser extensions [57]. In their system, extensions are written in Fine, a high-level, type safe programming language [115]. Web application security policies are written in Datalog and are verified against the browser extensions.

Our approach is similar. For programs written in a subset of JavaScript, we develop a tool to automatically translate these programs to ML. Using these translations, we can type check the JavaScript programs, assuming a sound translation, and prove basic properties about the programs. We show several example translations of real browser extensions, and show how type checking can be used to prove interesting properties of these extensions.

This has the potential to provide much more flexibility than the current systems for JavaScript because it is static but potentially more expressive than current verifiers. While it currently has many limitations, it shows a new, novel, promising path for JavaScript verification by translating JavaScript into other forms.

We show a novel translation of JavaScript into a statically refined, dependent, typed language. We also develop a library for modeling JavaScript behavior, including the DOM. We show how this translation can be used by a verifier for enforcing JavaScript policies on browser extensions.

Chapter 2

Background and Related Work

In this chapter, we discuss general background and related work to security policies for the web. In each succeeding chapter, we include work that relates directly to that particular subject.

While the goal of web application security policies applies to the larger problem of content injection, much of the related work relates to the more specific problem of cross-site scripting. Here we discuss much of the related research to these problems, with a focus on XSS research.

A recent study by Scholte, et al. shows that despite all of the work on and tools built for XSS, the presence of XSS vulnerabilities in web applications has not improved recently. [109] That is, both the quantity of XSS vulnerabilities, the complexity, or the presence of vulnerabilities in old versus new web applications has not significantly changed over the past 4 or 5 years. This implies that the current infrastructure and tools are not sufficiently addressing the current set of problems.

XSS Analysis and Defense Much of the research on cross-site scripting vulnerabilities has focused on finding XSS flaws in web applications, specifically on server-side code [140, 83, 82, 138, 77, 72, 99, 92, 20] but also more recently on JavaScript code [105, 106, 21, 58]. These works have underscored the two main causes of XSS vulnerabilities: *identifying untrusted data* at output and *errors in sanitization* by applications. There have been three kinds of defenses: purely server-side, purely browser-based, and those involving both client and server collaboration.

BLUEPRINT [120], SCRIPTGARD [107] and XSS-GUARD [28] are two server-side solutions that have provided insight into context sensitive sanitization. In particular, BLUEPRINT provides a deeper model of the web browser and points to paths between the browser components may vary across browsers. We discuss BLUEPRINT in particular detail in our proposed work in Section 1.2.

Purely browser-based solutions, such as XSSAuditor, and client-only solutions, such as DSI [98], have been implemented in modern browsers. These mechanisms are useful in nullifying common attack scenarios by observing HTTP requests and intercepting HTTP responses during the browser's parsing. However, they do not address the problem of sep-

arating what is untrusted from trusted data, as pointed out by Barth et al. [26]. Other language-based solutions for customizable XSS security policies are also an area of active research [95].

Cross-site scripting attacks have also been shown to result from unsafe parsing of CSS [71], optimistic content-sniffing algorithms in browsers [22], and from vulnerabilities in extensions [21, 23]. Failure to isolate mashups and advertisements may also result in code injection vulnerabilities, but typically the safety properties that these attacks violate are treated as a separate class from XSS vulnerabilities. These violated properties include isolation of principles in web browser primitives [130], authority safety [91] and statically verified containment [53].

Client-server collaborative defenses have been investigated in BEEP [76], DSI and NonceSpaces [61]. In these proposals, the server is responsible for identifying untrusted data which it reports to the browser, and the browser is modified to ensure that XSS attacks can not result from parsing the untrusted data. While these proposals are encouraging, they require updates in browser implementations as well as server-side code. The closest practical implementation of such client-server defense architecture is the recent *content security policy* specification [113]. We discuss the limitations of CSP in greater detail in Section 1.2.

Correctness of Sanitization While several systems have analyzed server-side code, the SANER [20] system empirically showed that custom sanitization routines in web applications can be error-prone. FLAX [106] and KUDZU [105] empirically showed that sanitization errors are not uncommon in client-side JavaScript code. While these works highlight examples, the complexity of the sanitization process remained unexplained. Our observation is that sanitization is pervasively used in emerging web frameworks as well as large, security-conscious applications. We discuss whether applications should use sanitization for defense in light of previous bugs.

Among server-side defenses, BLUEPRINT provided a sanitization-free strategy for preventing cross-site scripting attacks, which involved the explicit construction of the intended parse tree in the browser via JavaScript. We observe that sanitization-free mechanisms stand in contrast to whitelist-based canonicalization sanitization which is what is generally implemented in emerging frameworks, the security of which has neither been fundamentally broken nor proven. Research on string analysis and other automata-based verification systems is currently active, and this research is directly relevant to these questions [79, 105, 68].

Techniques for Separating Untrusted Content Taint-tracking based techniques aimed to address the problem of identifying and separating untrusted data from HTML output to ensure that untrusted data gets sanitized before it is output [99, 140, 77, 128, 34, 110]. Challenges in implementing taint-analysis as well as performance overheads have precluded their use in deployed web applications. Security-typed languages and type-systems offer another mechanism to ensure the robust isolation of untrusted data from HTML code output [35, 117, 103, 111]. The generality of type systems allows for creating a finer separation

between untrusted inputs, a property we motivate our empirical analysis. HTML templating engines, such as those studied in this work, offer a different model in which they coerce developers into explicitly specifying trusted content. This offers a fail-closed design and has seen adoption in practice because of its ease of use.

Chapter 3

Evaluating Implicit Policy Mechanisms in Web Frameworks

3.1 Introduction

Cross-site scripting (XSS) attacks are an unrelenting threat to existing and emerging web applications. Major web services such as Google Analytics, Facebook and Twitter have had XSS issues in recent years despite intense research on the subject [108, 74, 124]. Though XSS mitigation and analysis techniques have enjoyed intense focus [28, 98, 103, 80, 129, 18, 120, 106, 105, 140, 82, 138, 77, 72, 99, 92, 20, 26], research has paid little or no attention to a promising sets of tools for solving the XSS riddle—*web application frameworks*—which are gaining wide adoption [62, 37, 43, 112, 104, 48, 144, 141, 75, 93, 119]. Many of these frameworks claim that their sanitization abstractions can be used to make web applications secure against XSS [50, 141]. Though possible in principle, this chapter 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 to provide comprehensive XSS defense.

Researchers have proposed defenses ranging from purely server-side to browser-based or both [28, 98, 103, 80, 129, 18]. However, *sanitization* or *filtering*, the practice of encoding or eliminating dangerous constructs in untrusted data, remains the industry-standard defense strategy [100]. At present, each web application needs to implement XSS sanitization manually, which is prone to errors [20, 107]. 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 must un-

This chapter was previously published at the 2011 European Symposium on Research in Computer Security [135].

derstand the assumptions web frameworks build on and the security of their underlying sanitization mechanisms.

XSS 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 fully explained in prior XSS 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 XSS 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 XSS vulnerabilities in client-side JavaScript code.

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

- **Context Expressiveness**. Untrusted data needs to be sanitized differently based on its *context* in the HTML document. For example, the sanitization requirements of a URI attribute are different from those of an HTML tag. *Do web frameworks provide sanitizers for different contexts that applications commonly use in practice?*

- **Auto-sanitization and Context-sensitivity**. Applying sanitizers in code automatically, which we term *auto-sanitization*, shifts the burden of ensuring safety against XSS from developers to frameworks. However, a sanitizer that may be safe for use in one context may be unsafe for use in another. Therefore, to achieve security, auto-sanitization must be *context-sensitive*; otherwise, as we explain in Section 3.3, it may provide a false sense of security. *To what extent do modern web frameworks offer context-sensitive auto-sanitization?*

- **Security of dynamic client-side evaluation**. AJAX applications have significant client-side code components, such as in JavaScript. There are numerous subtleties in XSS sanitization because client-side code may read values from the DOM. *Do frameworks support complete mediation on DOM accesses in client-side code?*

Contributions and Approach We explain the challenges inherent in XSS sanitization. We present a novel model of the web browser's parsing internals in sufficient detail to explain the subtleties of XSS sanitization. Our model is the first to comprehensively conceptualize the difficulties of sanitization. Our browser model includes details of the sub-languages supported by HTML5, their internal interactions, and the transductions browsers introduce on content. We provide examples of XSS scenarios that result.

This chapter is a first step towards initiating research on secure web frameworks. It systematically identifies the features and pitfalls in XSS sanitization abstractions of today's

web frameworks and the challenges a secure framework must address. We compare existing abstractions in frameworks to the requirements of web applications, which we derive by an empirical analysis. We study 14 mature, commercially used web frameworks and 8 popular open-source web applications. We establish whether the applications we study could be migrated to use the abstractions of today's web frameworks. We quantify the security of the abstractions in frameworks and clarify the liability developers will continue to take even if they were to migrate their applications to today's frameworks. We provide the first in-depth study of the gap between the sanitization abstractions provided by web frameworks and what web applications require for safety against XSS. We conclude that though web frameworks have the potential to secure applications against XSS, most existing frameworks fall short of achieving this goal.

3.2 A Systematic Browser Model for XSS

We formulate XSS with a comprehensive model of the browser's parsing behavior in Section 3.2. We discuss the challenges and subtleties XSS sanitization must address in Section 3.2, and how web frameworks could offer a potential solution in Section 3.2. We outline our evaluation objectives and formulate the dimensions along which we empirically measure the security of web frameworks in Section 3.2.

Problem Formulation: XSS Explained

Web applications mix control data (code) and content in their output, generated by server-side code, which is consumed as client-side code 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, an XSS attack results. A canonical example of an XSS 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.

XSS 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 XSS much more complicated. Previous research has indicated that this problem is complex, but we are not aware of an in-depth, systematic problem formulation.

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 [143], a formal model has not been built to fully explore its complexity. We show this model in Figure 3.1. Abstractly, the browser can be viewed as a collection of HTML-related sub-

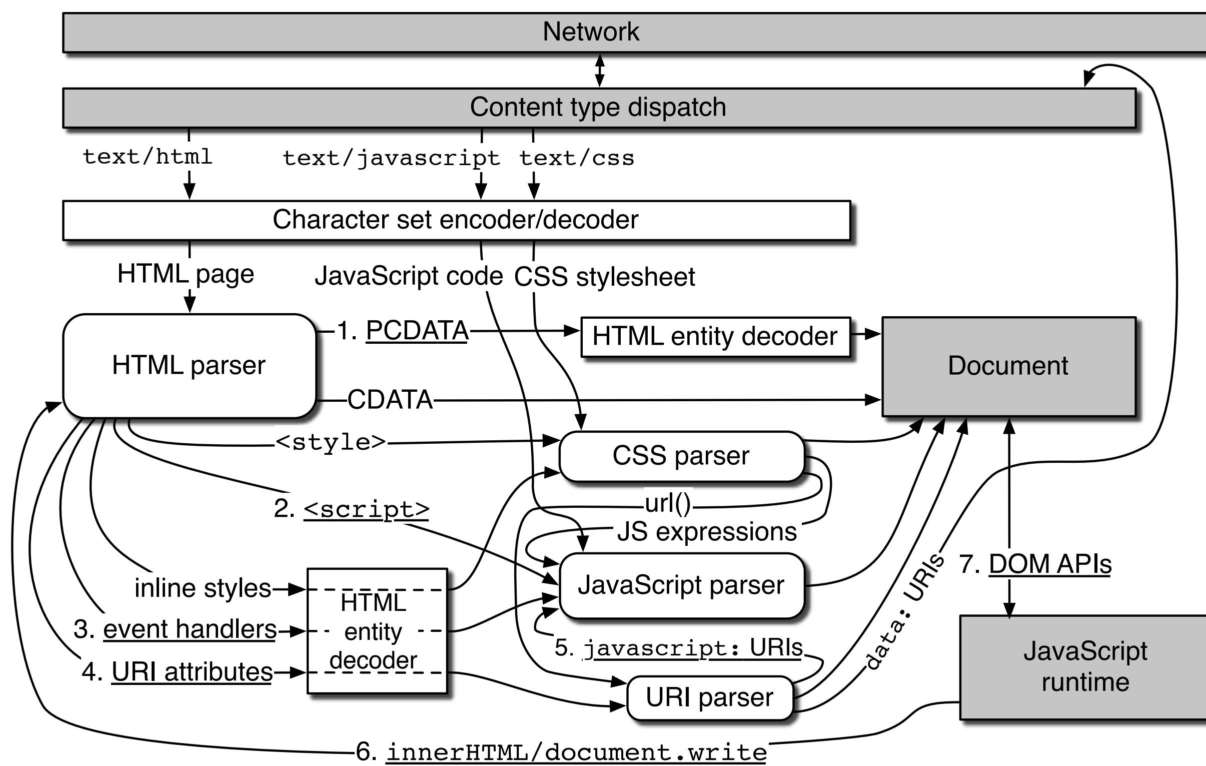

Figure 3.1: Flow of Data in our Browser Model. Certain contexts such as `PCDATA` and `CDATA` directly refer to parser states in the HTML 5 specification. We refer to the numbered and underlined edges during our discussion in the text.

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 3.1). 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 3.1). 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 preprocessing 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 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 XSS 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 XSS attacks when is placed inside the body of an HTML tag, or, in the PCDATA (edge 1) parsing context, as defined by HTML5 [67]. 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 [63].

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 3.1.

Sanitizing for nested contexts adds its own complexity. Consider an embedding of an untrusted string 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 3.1. 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 `"` 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 call such implicit conversions *browser transductions*. Full details of the transductions are available in Appendix A.

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 3.1), 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 XSS or client-side code injection [123]. In contrast to Challenges 2 and 3, such vulnerabilities are caused not by a lack of understanding the browser, but of frameworks not understanding application behavior.

Another key observation is that browser transductions along the edges of Figure 3.1 vary from one edge to another, as detailed in Appendix A. This mismatch can cause XSS vulnerabilities. During our evaluation, we found one such bug (Section 3.3). We speculate that JavaScript-heavy web applications are likely to have such vulnerabilities.

Challenge 5: Character-set Issues Successfully sanitizing a string at the server side implicitly requires that the sanitizer and the browser are using the same character set while working with the string. A common source of XSS 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 XSS vulnerabilities [125].

Challenge 6: MIME-based XSS, Universal XSS, and Mashup Confinement Browser quirks, especially in interpreting content or MIME types [22], contribute their own share of

XSS vulnerabilities. Similarly, bugs in browser implementations, such as capability leaks [53] and parsing inconsistencies [137], or in browser extensions [23] 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.

The Role of Web frameworks

Web application development frameworks provide components to enable typical work flows in web application development. These frameworks can abstract away repetitive and complex tasks, freeing the developer to concentrate on his particular scenario. Consider session management, a common feature that is non-trivial to implement securely. Most web application frameworks automate session management, hiding this complexity from the developer. Similarly, web application frameworks can streamline and hide the complexity of XSS sanitization from the developer. In fact, increased security is often touted as a major benefit of switching to web application frameworks [50, 141].

Frameworks can either provide XSS sanitization routines in a library or they can automatically add appropriate sanitization code to a web application. We term the latter approach *auto-sanitization*. In the absence of auto-sanitization, the burden of calling the sanitizers is on the developer, which we have seen is an error-prone requirement. On the other hand, auto-sanitization, if incorrectly implemented, can give a false sense of security because a developer may defer all sanitization to this mechanism.

Analysis Objectives

In theory, use of a web application framework should free the developer from the complexities of XSS sanitization as discussed earlier and illustrated in Figure 3.1. If true, this requires the framework to grapple with all these complexities instead. We abstract the most important challenges into the following three dimensions:

- **Context Expressiveness and Sanitizer Correctness.** As we detailed in Challenge 1, sanitization requirements change based on the context of the untrusted data. We are interested in investigating the set of contexts in which untrusted data is used by web 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 3.2 make manually developing *correct* sanitizers a non-starter. Instead, we ask, *do web frameworks provide correct sanitizers for different contexts that web 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 XSS attacks. Applying sanitizers in code automatically, which we term *auto-sanitization*, shifts the burden of ensuring safety against XSS 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 it should be performed. 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?*

- **Security of client-side code evaluation.** Much of the research on XSS has focused on the subtleties of parsing in HTML contexts across browsers. But AJAX web applications have significant client-side code components, such as in JavaScript. There are numerous subtleties in XSS sanitization because client-side code may read values from the DOM. Sanitization performed on the server-side may be "undone" during the browser's parsing of content into the DOM (Challenge 3 and Challenge 4). *Do frameworks support complete mediation on DOM accesses in client-side code?*

In this study, we focus solely on XSS sanitization features in web frameworks and ignore all other framework features. We also do not include purely client-side frameworks such as jQuery [3] because these do not provide XSS protection mechanisms. Additionally, untrusted data used in these libraries also needs server-side sanitization.

3.3 Analysis of Web Frameworks and Applications

In this section, we empirically analyze web frameworks and the sanitization abstractions they provide. We show 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 3.3. We identify the extent to which it is available, the pitfalls of its implementation, and whether developers can blindly trust this mechanism if they migrate to or develop applications on existing auto-sanitizing frameworks. We then evaluate the support for dynamic code evaluation via JavaScript in frameworks in Section 3.3. In the previous section, we identified subtleties in the browser's DOM interface. In Section 3.3, we discuss whether applications adequately understand it to prevent XSS bugs.

Frameworks may not provide auto-sanitization, but instead may provide sanitizers that 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, in Section 3.3. We also compare it to the requirements of the applications we study today to evaluate whether this expressiveness is enough for real-world applications.

Finally, we evaluate frameworks' assumptions regarding correctness of sanitization and compare these to the sanitization practices in security-conscious 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 parsed the outputs using an HTML 5-compliant browser to determine the contexts for which they sanitize. Due to space constraints, this chapter focuses solely on the results of our empirical analysis. A technical report provides the full details of the techniques employed [134].

Auto-Sanitization: Features and Pitfalls

Auto-sanitization is a feature that shifts the burden of ensuring safety against XSS 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 3.1, 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 3.2 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. In those that don’t, developers must deal with the challenges of selecting where to apply built-in or custom sanitizers. Recent studies have shown that this manual process is prone to errors, even in security-audited applications [107, 52]. We also observed instances of this phenomenon in our analysis. The following example is from a Django application called GRAMPS.

Example 1

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

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 [17]	•		Request Reception
Ruby	xss_terminate Rails plugin [139]	•		Database Insertion
Python	Django	•		Template Processing
Java	GWT SafeHtml	•	•	Template Processing
C++	Ctemplate	•	•	Template Processing
Language-neutral	ClearSilver	•	•	Template Processing

Table 3.1: 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.

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 3.2. In particular, this sanitizer fails to prevent XSS 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 XSS attacks. Django employs context-insensitive auto-sanitization, i.e., it applies the same sanitizer (`escape`) irrespective of the output context. `escape`, which does an HTML entity encode, is 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 XSS attacks. Auto-sanitization support in Rails [139], .NET (request validation [17]) and CodeIgniter are all context-insensitive and have 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 XSS 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 [49]. We statically correlated the automatically applied sanitizer to the context of the data; the results are in Table 3.2. The mechanism protects between 66.4% and 85.2% of the output sinks, but conversely permits XSS 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. An auto-sanitization mechanism that requires developers to sanitize diligently is self-defeating. Developers should be aware of this responsibility when building on such a mechanism.

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 3.2: 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.

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:

Example 2

```
{{%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>
```

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 `{{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.

Security of Client-side Code Evaluation

In Section 3.2, we identified subtleties of dynamic evaluation of HTML via JavaScript's DOM API (Challenge 4). The browser applies different transductions depending on the DOM interface used (Challenge 3 and listed in Appendix A). Given the complexity of sanitizing dynamic evaluation, we believe web frameworks should provide support for this important class of XSS attack vectors too. Ideally, a web framework could incorporate knowledge of these subtleties, and provide automatic sanitization support during JavaScript code execution.

Support in web frameworks The frameworks we studied do not support sanitization of dynamic flows. Four frameworks support sanitization of untrusted data used in a JavaScript string or number context (Table 3.3). This support is only *static*: it can ensure that untrusted data doesn't escape out during the parsing by the browser, but such sanitization can't offer any safety during dynamic code evaluation, given that dynamic code evaluation can *undo* previously applied transductions (Challenge 4).

Context-insensitivity issues with auto-sanitization also extend to JavaScript code. For example, Django uses the context-insensitive HTML-escape sanitizer even in JavaScript string contexts. Dangerous characters (e.g., `\n`,`\r`,`;`) can still break out of the JavaScript string literal context. For example, in the Malaysia Crime Application (authored in Django), the `crime.icon` variable is incorrectly auto-sanitized with HTML-entity encoding and is an argument to a JavaScript function call.

Example 3

```
map.addOverlay(new GMarker(point, {{ crime.icon }}))
```

Awareness in Web Applications DOM-based XSS is a serious problem in web applications [106, 105]. Recent incidents in large applications, such as vulnerabilities in Google optimizer [102] scripts and Twitter [123], show that this is a continuing problem. This suggests that web applications are not fully aware of the subtleties of the DOM API and dynamic code evaluation constructs (Challenge 3 and 4 in Section 3.2).

To illustrate this, we present a real-world example from one of the applications we evaluated, `phpBB3`, showing how these subtleties may be misunderstood by developers.

Example 4

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

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 like <script>, the encoding converts it to <script>. 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 3.1) 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 in Appendix A.

The prevalence of DOM-based XSS vulnerabilities and the lack of framework support suggest that this is a challenge for web applications and web frameworks alike. Libraries such as Caja and ADsafe model JavaScript and DOM manipulation but target isolation-based protection such as authority safety, not DOM-based XSS [31, 41]. Protection for this class of XSS requires further research.

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 3.3 presents the expressiveness of web frameworks we study and Table 3.4 presents the expressiveness required by our subject web applications. The key insights are:

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 [14, 93]	•	•				
	Template Toolkit [119]	•	•				
	Jifty [75]	•	•				
PHP	CakePHP [32]	•	•				
	Smarty Template Engine [112]	•	•		•		
	Yii [141, 70]	•	•				
	Zend Framework [144]	•	•				
	CodeIgniter [39, 38]	•	•				
VB, C#, C++, F#	ASP.NET [16]	•	•				
Ruby	Rails [104]	•	•				
Python	Django [48]	•	•	•	•		
Java	GWT SafeHtml [62]	•	•	•			
C++	Ctemplate [43]	•	•	•	•	•	•
Language-neutral	ClearSilver [37]	•	•	•	•		•

Table 3.3: Sanitizers provided by languages and/or frameworks. For frameworks, we also include sanitizers provided by standard packages or modules for the language.

- 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 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 [41] or Google's Caja [31].

- 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 [70], 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,

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

Table 3.4: The web applications we study and the contexts for which they sanitize.

untrusted content within a particular context is always sanitized the same way. Our evaluation of web applications strongly invalidates this assumption, showing that applications often sanitize data occurring in the same context differently based on other attributes of the data.

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

Expressiveness of Contexts and Sub-Context Variance 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 3.4 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 3.3, where most frameworks support a much more limited set of contexts with built-in sanitizers.

More surprisingly, we find that applications often employ more than one sanitizer for each context. That is, an abstraction that ties a single sanitizer to a given context may be insufficient. We term this variation in sanitization across code paths *sub-context variance*. Sub-context variance evidence suggests that directly migrating web applications to web frameworks' (auto-) sanitization support may not be directly possible given that even context-sensitive web frameworks rigidly apply one sanitizer for a given context.

Sub-context variance is particularly common in the form of *role-based* sanitization, where the application applies different sanitizers based on the privilege of the user. We found that it is common to have a policy in which the site administrator's content is subject to

no sanitization (by design). Examples include phpBB, WordPress, Drupal. For such simple policies, there are legitimate code paths that have no sanitization requirements. To illustrate this, we present a real-world example from the popular WordPress application which employs different sanitization along different code paths.

Example 5

> WordPress, the popular blogging application, groups users into *roles*. A user in the author role can create a new post on the blog with most non-code tags permitted. An anonymous commenter, on the other hand, can only use a small number of text formatting tags. In particular, the latter cannot insert images in comments while an author can insert images in his post. Note that neither can insert `<script>` tags, or any other active content. In both cases, untrusted input flows into HTML tag context, but the sanitizer applied changes as a function of the user role.

Most auto-sanitizing frameworks do not support such rich abstractions to support auto-sanitization specifications at a sub-context granularity. Nearly all sanitization libraries (not part of web frameworks) are customizable. However, their connection to special role-based sanitization (or similar cases) are not supported presently. We believe that web frameworks can fill this gap. Only 1 framework, Yii, provides the flexibility to handle such customizations using the HTMLPurifier sanitization library. Unfortunately, Yii only provides this flexibility for the HTML tag context.

Enabling Reasoning of Sanitizer Correctness

Prior research on web applications has shown that developing sanitization functions, especially custom sanitizers, is tricky and prone to errors [20]. We investigate how the sanitizers in web frameworks handle this issue. We compare the structure of the sanitizers used in frameworks to the structure we observe in our subject applications and characterize the ground assumptions that developers should be aware of.

Blacklists vs. Whitelists We find that most web frameworks structure their sanitizers as a *declarative-style whitelist* of code constructs explicitly allowed in untrusted content. For instance, one sanitization library employed in the Yii is HTML-Purifier [70], which permits a declarative list of HTML elements like event attributes of special tags in untrusted content. All of the web applications we studied also employ this whitelisting mechanism, such as the KSES library used in WordPress [81]. Such sanitizers assume that the whitelist is only contains a well understood and safe subset of the language specification, and does not permit any unsafe structures.

In contrast, we find that only 1 subject web framework, viz. CodeIgniter, employs a blacklist-based sanitization approach. Even if one verifies that all the elements on a blacklist conform to an unsafe subset of the language specification, the sanitizer may still allow unsafe

parts of the language. For example, CodeIgniter's `xss_clean` function removes a blacklist of potentially dangerous strings like `document.cookie` that may appear in any context. Even if it removes *all* references to `document.cookie`, there still may be other ways for attacker code to reference cookies, such as via `document['cookie']`.

Correctness of the sanitizers used is fundamental to the safety of the sanitization based strategy used in web frameworks. Based on the above examples, we claim that it is easier to verify that a whitelist policy is safe and recommend frameworks adopt such a strategy.

HTML Canonicalization Essential to the safety of sanitization-based defense is that the user's browser parse the untrusted string in a manner consistent with the parsing applied by the sanitizer. For instance, if the context-determination in the frameworks differs from the actual parsing in the browser, the wrong sanitizer could be applied by the framework.

We observe that frameworks employ a *canonicalization* strategy to ensure this property; the web frameworks identify a 'canonical' subset of HTML-related languages into which all application output is generated. The assumption they rely on is that this canonical form parses the same way across major web browsers. We point out explicitly that these assumptions are not systematically verified today and, therefore, framework outputs may still be susceptible to XSS attacks. For example, a recent XSS vulnerability in the HTML Purifier library (used in Yii) was traced back to "quirks in Internet Explorer's parsing of string-like expressions in CSS [69]."

Finally, we point out that sanitization-based defense isn't the only alternative—proposals for *sanitization-free* defenses, such as DSI [98], BLUEPRINT [120] and the Content Security Policy [113] have been presented. Future frameworks could consider these. Verifying the safety of the whitelist-based canonicalization strategy and its assumptions also deserves research attention.

3.4 Related Work

XSS Analysis and Defense Much of the research on cross-site scripting vulnerabilities has focused on finding XSS flaws in web applications, specifically on server-side code [140, 83, 82, 138, 77, 72, 99, 92, 20] but also more recently on JavaScript code [105, 106, 21, 58]. These works have underscored the two main causes of XSS vulnerabilities: *identifying untrusted data* at output and *errors in sanitization* by applications. There have been three kinds of defenses: purely server-side, purely browser-based, and those involving both client and server collaboration.

BLUEPRINT [120], SCRIPTGARD [107] and XSS-GUARD [28] are three server-side solutions that have provided insight into context-sensitive sanitization. In particular, BLUEPRINT provides a deeper model of the web browser and points out that browsers differ in how the various components communicate with one another. The browser model detailed in this work builds upon BLUEPRINT's model and more closely upon SCRIPTGARD's formaliza-

tion [107]. We provide additional details in our model to demystify the browser's parsing behavior and explain subtleties in sanitization that the prior work did not address.

Purely browser-based solutions, such as XSSAuditor, are implemented in modern browsers. These mechanisms are useful in nullifying common attack scenarios by observing HTTP requests and intercepting HTTP responses during the browser's parsing. However, they do not address the problem of separating untrusted from trusted data, as pointed out by Barth et al. [26].

BEEP, DSI and NonceSpaces investigated client-server collaborative defenses. In these proposals, the server is responsible for identifying untrusted data, which it reports to the browser, and the browser ensures that XSS attacks can not result from parsing the untrusted data. While these proposals are encouraging, they require browser and server modifications. The closest practical implementation of such client-server defense architecture is the recent *content security policy* specification [113].

Correctness of Sanitization While several systems have analyzed server-side code, the SANER [20] system empirically showed that custom sanitization routines in web applications can be error-prone. FLAX [106] and KUDZU [105] empirically showed that sanitization errors are not uncommon in client-side JavaScript code. While these works highlight examples, the complexity of the sanitization process remained unexplained. Our observation is that sanitization is pervasively used in emerging web frameworks as well as large, security-conscious applications. We discuss whether applications should use sanitization for defense in light of previous bugs.

Techniques for Separating Untrusted Content Taint-tracking based techniques [99, 140, 77, 128, 34, 110] as well as security-typed languages [35, 117, 103, 111] aim to address the problem of identifying and separating untrusted data from HTML output to ensure that untrusted data gets sanitized before it is output. Web templating frameworks, some of which are studied in this work, offer a different model in which they coerce developers into explicitly specifying trusted content. This offers a fail-closed design and has seen adoption in practice because of its ease of use.

3.5 Conclusions and Future Work

We study the sanitization abstractions provided in 14 web application development frameworks. We find that frameworks often fail to comprehensively address the subtleties of XSS sanitization. We also analyze 8 web applications, comparing the sanitization requirements of the applications against the abstractions provided by the frameworks. Through real-world examples, we quantify the gap between what frameworks provide and what applications require.

Auto-sanitization Support and Context Sensitivity Automatic sanitization is a step in the right direction. For correctness, auto-sanitization needs to be context-sensitive: context-insensitive sanitization can provide a false sense of security. Our application study finds that applications do, in fact, need to emit untrusted data in multiple contexts. However, the total number of contexts used by applications in our study is limited, suggesting that frameworks only need to support a useful subset of contexts.

Security of Client-side Code Evaluation DOM-based XSS is a serious challenge in web applications, but no framework supports sanitization for dynamic evaluation on the client. Application developers must be particularly alert when using the DOM API. Of particular relevance to XSS sanitization is the possibility of the browser "undoing" server-side sanitization, making the application vulnerable to DOM-based XSS.

Context Expressiveness and Sanitizer Correctness Some frameworks offer sanitization primitives as library functions the developer can invoke. We find that most frameworks do not provide sufficiently expressive sanitizers, i.e., the sanitizers provided do not support all the contexts that applications use. For instance, applications emit untrusted data into URI attribute and JavaScript literal contexts, but most of the frameworks we study do not provide sanitizers for these contexts. As a result, application developers must implement these security-critical sanitizers themselves, a tedious and error-prone exercise. We also find that sub-context variance, such as role-based sanitizer selection, is common. Only one of the frameworks we examined provides any support for this pattern, and its support is limited.

Finally, our study identifies the set of assumptions fundamental to frameworks. Namely, frameworks assume that their sanitizers can be verified for correctness, and that HTML can be canonicalized to a single, standard form. Developers need to be aware of these assumptions before adopting a framework.

Future Directions As we outline in this work, the browser's parsing and transformation of the web content is complex. If we develop a formal abstract model of the web browser's behavior for HTML 5, sanitizers can be automatically checked for correctness. Our browser model is a first step in this direction. We identify that parts of the web browser are either transducers or language recognizers. There have been practical guides for dealing with these issues, but a formal model of the semantics of browsers could illuminate all of the intricacies of the browser [143]. Verification techniques and tools for checking correctness properties of web code is an active area of research.

If one can show the correctness of a framework's sanitizers, we can prove the security and correctness for code generated from it. Though existing auto-sanitization mechanisms are weak today, they can be improved. Google AutoEscape is one attempt at this type of complete sanitization but is currently limited to a fairly restrictive templating language [43]. If these abstractions can be extended to richer web languages, it would provide a basis to

build web applications secure from XSS from the ground up—an important future direction for research.

Acknowledgments Thanks to my co-authors of this work, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song [135], for working with me and giving their permission for this work to appear in this thesis.

Chapter 4

Evaluating the Effectiveness of Content Security Policy

4.1 Introduction

Content injection attacks, and in particular cross-site scripting (XSS) attacks, are a significant threat to web applications [136, 5]. These attacks violate the integrity of web applications, steal from users and companies, and erode privacy on the web. Researchers have focused a great deal of effort at preventing these attacks, ranging from static detection to dynamic checking of content.

The classic approach to stopping content cross-site scripting attacks is by the careful placement of *sanitizers*. Sanitizing, or filtering, is the removal of potentially harmful content or structure from untrusted data. Sanitization places a heavy burden on developers; they must identify where untrusted data appears in their application, what that data is allowed to do, and the context the data appears in on the final page. Unfortunately, these are not straightforward processes. Sanitization is a very brittle process, prone to error. Besides often missing untrusted data in their application, it is not always obvious what sanitization method to apply in what context [135, 107].

In response, several alternatives have been proposed in the literature, most notably, BEEP [76], BLUEPRINT [120], and Content Security Policy (CSP) [113]. These proposals suggest very different mechanisms for preventing XSS and, in some of the cases, more general content injection. Previously, these have been viewed separate proposals with different approaches to the same problem. However, we identify these proposals as instances of a more general notion of *client-side HTML security policies.*

This chapter was previously published at the 2011 USENIX Workshop on Hot Topics in Security [133].

We argue that HTML security policies are superior to sanitization and should be the core defensive mechanism against content injection in future web applications. However, we also argue and show that the current proposals for HTML security policies fall short of their ultimate design goals. We argue that HTML security policies should be at the core of web application security, but much research still needs to be done in building successful HTML security policy systems. We put forth several suggestions for future research, but generally leave next steps as an open problem.

HTML Security Policy Systems We focus on BEEP, BLUEPRINT, and Content Security Policy. We examine these systems to understand their respective power and limitations. While all three provide working systems for expressing and implementing HTML security policies, CSP is of particular importance because it is deployed in the Firefox [12], Safari [13], and Chrome [11] web browsers. We evaluate the efficacy of all three, including the first empirical evaluation of CSP on real applications by retrofitting Bugzilla [1] and HotCRP [2] to use CSP. We conclude that none of these systems solves the content injection or XSS problems sufficiently:

- BEEP has serious limitations in dealing with dynamic script generation. This is especially problematic in today's web frameworks that support and encourage templating across all types of code. Additionally, it does not provide any support for non-XSS content injection.
- BLUEPRINT has significant performance problems. These are not superficial; there are inherent to its approach in avoiding use of the browser's parser.
- CSP can cause significant performance problems in large applications because of the restrictions on code it enforces. Additionally, it does not fit well into web framework the programming models.

Proposals We argue that going forward, web frameworks should use HTML security policies instead of sanitization as the basis of their protection against content injection. However, the security community needs to decide on a set of requirements that HTML security policies should hold going forward. In the end, new HTML security policy systems, and perhaps new browser primitives, need to be developed by the research community to solve the content injection problem.

4.2 HTML Security Policies

Content injection occurs when untrusted user input changes the intended parse tree of a document [28, 98]. In the case of a web page, this happens when a user input is placed on a page, and the user input contains control structures (such as HTML tags or JavaScript

code) that the developer did not intend to be present. By definition, this allows an attacker to modify the behavior of the page in unexpected ways, changing the developer's intended semantics. XSS is a specific type of content injection where the attacker modifies the document's structure to place a script on the page.

Web developers generally use sanitization to defend from content injection, but sanitization can be difficult to get right. Thus, researchers have proposed several other mechanisms for protecting web applications from content injection. Three well known proposals, BEEP, BLUEPRINT, and Content Security Policy (CSP), are instances of HTML security policy systems. HTML security policy systems provide mechanisms for an application to specify how a document is parsed and interpreted by the browser on the client, in contrast to the syntactic approach of sanitization on the server, where unwanted syntax is removed before it reaches the client.

In HTML security policy systems, a policy is provided to the browser when a web page is loaded, and as the browser renders the page, it enforces that the page matches the HTML security policy. This is particularly useful in the context of a web server providing dynamically generated content where a policy is hard to enforce statically (such as a page containing user comments, a news aggregator, etc.).

The advantages of HTML security policy systems over sanitization are several fold. The key improvement is that developers do not need to search for the precise place in code that untrusted data may appear. In sanitization, these places are hard to find, and even when found, it is not necessarily clear what sanitizer to apply (i.e. does the untrusted data appear between HTML tags, or is it part of an attribute, or another context entirely?) [135]. In comparison, one of the goals of HTML security policy systems is to specify allowed behavior, not to limit the syntax at these precise points.

Additionally, in HTML security policy systems, there is an explicit policy to enforce, in contrast to the ad hoc application of sanitizers. This suggests that HTML security policies are easier to audit than sanitization. A developer can check a policy against their security model, rather than searching an application for sanitizers and building a policy from that.

Because of these properties, and the unreliability of and difficulty in using sanitizers, we argue that HTML security policy systems should be used going forward in web applications to help solve the content injection problem instead of sanitization. However, the current set of available HTML security policy systems have several major problems in their design.

Existing Policy Systems

While there are many different HTML security policy systems, we focus on three of the most cited, BEEP [76], BLUEPRINT [120] and CSP [113]. These three systems take two very different approaches to the same problem: how to stop untrusted content from injecting additional structure into a web page. They are all particularly concerned with cross-site scripting attacks.

BEEP BEEP focuses on XSS instead of the more general content injection problem. BEEP implements a whitelist of trusted scripts that may execute and rejects the execution of any script not on the whitelist. Thus, only trusted, developer-built scripts may execute on the client, and any injected scripts will fail to do so.

Unfortunately, BEEP's handling of dynamically generated scripts does not match the web framework model. BEEP requires that the hash of a script be statically determined or that the script is added by a trusted script. By definition, if the script is generated dynamically, its hash cannot be determined statically. One can imagine a system with scripts that add dynamically generated scripts, but this is very different from how web applications and frameworks currently handle code generation.

Additionally, BEEP does not handle content injection other than XSS, which have recently been seen in real sites [121]. Also, attacks on BEEP have been developed similar to return-to-libc attacks [19]. While more complex than traditional XSS, the existence of such attacks is cause for concern.

BLUEPRINT BLUEPRINT presents a system for parsing document content using a trusted, cross-platform, JavaScript parser, rather than browsers' built in parsers. The authors view HTML parsers in different browsers as untrustworthy because of browser quirks and view the cross-site scripting problem as fundamentally arising from this. Their approach provides the browser with a "blueprint" of the structure of the page, and a JavaScript library builds the page from the blueprint rather than trusting the browser's HTML parser.

This "blueprint" is an HTML security policy. The server parses the document itself and generates the structural blueprint of the document. This is communicated to the browser where it is used by the BLUEPRINT JavaScript library to build the document. The blueprint is a step-by-step instruction set for the structure of the page, and if any of the content violates this structure, it violates the policy and is removed.

One of the key assumptions of the authors is that server applications "know how to deal with untrusted content." Unfortunately, the authors make this assumption without defending it. There certainly are numerous cases of server applications that do not understand how to properly deal with untrusted content; this is the basis of SQL injection attacks [131]. A tool that could help well-intentioned developers stop *potentially* untrusted content would help to alleviate this.

Additionally, BLUEPRINT unfortunately suffers from several performance problems. In the original paper, the authors report 55% performance overhead in applying BLUEPRINT to Wordpress and 35% performance overhead in applying it to MediaWiki. Because of its very nature, BLUEPRINT cannot use the efficient parsing primitives of the browser; it relies entirely on building the document from the blueprint with the JavaScript parser.

Content Security Policy (CSP) To our knowledge, CSP is the first HTML security policy system to be implemented and released by one of the major browser vendors (in Firefox 4). CSP takes a different view of the browser than BLUEPRINT. Instead of "not trusting"

the browser's parsing decision, CSP implements a declarative policy that the browser then enforces on the application. CSP trusts the browser for enforcement, conceding that an application may be flawed.

CSP does this by developing a large set of properties that may be set on a *per page basis*. All trust is based on the page level by CSP properties that state trusted servers for scripts, images, and a variety of other content. This provides a strong fail-safe property. Because the entire page is covered by the policy, the policy will apply to all untrusted content wherever it appears on the page.

The types of properties that CSP provides include trusted servers for images, scripts, and other content, but it also includes one particularly important property. This is the `inline-scripts` property, which, by default, is disabled. When disabled, this means that the browser will not allow any scripts to execute within the page; the only scripts that may be run are ones included by the `src` attribute of a `<script>` tag. This is fundamentally how CSP prevents XSS attacks. Because no script content is allowed to run within the page, and the developer may set a small number of trusted servers for scripts to come from, an injection attack can add a script but it either will not run because it is inline, or it will not run because it will have a `src` pointing to an attacker's untrusted server.

CSP rules are a declarative way of specifying the dynamic semantics of a web page. CSP specifies a set of semantic rules on a per page basis. However, content injection is a syntactic problem where the abstract syntax tree of a document is modified by an untrusted source [114, 98]. It would be possible to keep adding semantic rules to CSP, but a new rule would be needed for each semantic consequence of all possible syntactic changes. Because of this, CSP only provides rules for stopping a small set of injection attacks, namely XSS and specific types of content (such as `<iframe>` tags whose `src` attribute points to an untrusted server). CSP does not stop general content injection, and for it do so would require an ever growing set of rules.

CSP's declarative, page-level, fail-safe architecture is enticing. However, it places severe restrictions on how web application pages can be structured. We evaluate how these restrictions affect real web applications.

4.3 Evaluating the Application of CSP

To evaluate the efficacy of CSP as an HTML security policy, we apply it to two popular, real world applications. We determine the HTML security policies necessary to stop injection attacks and apply the policies to the applications. We modify the applications to work with these policies. We evaluate the performance of these applications using CSP, and measure the effort of modifying these applications.

Methodology

The applications we experiment on are Bugzilla [1] and HotCRP [2]. Bugzilla is a web application for organizing software projects and filing and tracking bugs, used by many large companies and open source projects, including RedHat, Mozilla, Facebook, and Yahoo! [30]. HotCRP is a conference manager used for paper submission and review by several major conferences.

We retrofit the applications to execute properly with a known CSP policy that blocks XSS. As a manual process, we run the program and explore its execution space by clicking through the application. We modify the applications to correct any violations of the CSP policy by the applications. This does not provide completeness but we feel this most accurately represents what a developer would need to do to apply CSP to her application. While static and dynamic analysis tools have the potential to help, we are unaware of such tools for the Template Toolkit [119] language that Bugzilla is written in.

Application Modifications

The major part of transforming Bugzilla and HotCRP is converting inline JavaScript to external JavaScript files that are sourced by the HTML page. Because CSP policies are of page level granularity, it cannot reason about individual scripts on a page. Thus, in order to prevent XSS with CSP, it must reject inline scripts and only source scripts from trusted servers. The consequence of this is that completely trusted scripts must be moved to separate files.

Data Access In the implementations of Bugzilla and HotCRP, there are a variety of inline scripts that reference data and variables generated by the templating languages. such as configuration information or the number of search results. This data is not untrusted input. Unfortunately, when the scripts are segregated into separate files from the templated HTML, the variables and data can no longer be referenced by the templating language in the script source file. This is because the templating languages for both Bugzilla and HotCRP treat the individual page as a scoping closure; variables are not shared between separately sourced pages. We address this by creating additional hidden HTML structure and storing the necessary data in an attribute. Later, we extract this via additional JavaScript on the client.

DOM Manipulation DOM manipulation becomes necessary in a number of other contexts as well. Take as an example dynamically generated JavaScript. In HotCRP there are several scripts that interact with lists of papers. For each of the papers in the lists, there are inline JavaScript handlers for each entry, such as `onclick` handlers. Because CSP does not allow inline scripts, including handlers, these handlers must be inserted dynamically by JavaScript. Thus, for each paper entry in the list, we use PHP to generate a `span` with a predictable name, such as `name="paper-name-span"`, and also contains a

Page	No Inline JS	Async JS
index.php	$14.78\% \pm 4.5$	$-3.0\% \pm 4.25$
editsettings.php	$6.3\% \pm 4.7$	$5.1\% \pm 0.92$
enter_bug.cgi	$57.6\% \pm 2.5$	$44.2\% \pm 2.1$
show_bug.cgi	$51.5\% \pm 2.8$	$4.0\% \pm 3.0$

Table 4.1: Percent difference in performance between modified Bugzilla and original with 95% confidence intervals.

Page	No Inline JS	Async JS	JS Template
index.php	$45.3\% \pm 6.3$	$37.2\% \pm 5.0$	$27.9\% \pm 3.7$
search.php	$52.9\% \pm 5.4$	$50.4\% \pm 3.7$	$20.2\% \pm 3.9$
settings.php	$23.3\% \pm 2.7$	$16.1\% \pm 8.2$	—
paper.php	$61.1\% \pm 9.5$	$58.5\% \pm 8.7$	$19.1\% \pm 2.5$
contacts.php	$67.8\% \pm 4.8$	$35.5\% \pm 4.9$	—

Table 4.2: Percent difference in performance between modified HotCRP and original with 95% confidence intervals and jQuery Templating performance.

`data-paper-name` attribute. When the JavaScript executes, it searches for all elements with the name `paper-name-span`, and extracts the name of the element to add a handler to.

Additional Application Logic Another pattern we observe is the required movement of templating logic into JavaScript. Because the JavaScript is no longer inlined, the conditional branching in the templates no longer can affect it. Thus, it must replicate the same conditionals and checks dynamically. Using the same techniques as we discussed earlier, we replicate where necessary templating conditionals in JavaScript based on data passed in DOM element attributes. This adds additional performance costs to the execution, but also provides additional points of failure for the transformation.

Total Modifications Overall, the types of modifications to Bugzilla and HotCRP closely mirrored one another. This was particularly interesting given that they used two unrelated templating frameworks, Template Toolkit and PHP, respectively. In both cases, it was necessary to transfer data and variables to JavaScript through HTML structure, create significant additional DOM manipulations to build the application, and to move and duplicate application logic from the server to the client.

Our modifications were substantial, adding 1745 lines and deleting 1120 lines of code in Bugzilla and adding 1440 lines and deleting 210 lines of code in HotCRP. We also observed an increase in the number of `GET` requests and data transferred for Bugzilla and HotCRP.

Performance

The performance of the applications are affected in several different ways. During the application modification, we observe several particular modifications that relate to performance:

- **Additional script source files** In order to remove inline scripts from Bugzilla and HotCRP, we add a number of new script source files. When possible, we consolidate what were separate inline scripts into one source file, but this is not always possible. For example, if a script uses `document.write`, the script must be placed in a specific location. Additionally, many scripts are conditional on templating branching, and should only occur if specific elements exist or particular properties hold. These extra files have the potential to add latency to page load and execution.

- **New DOM manipulations** As discussed above, to transfer data and variables from the template to the script, we attach the data to HTML elements and use JavaScript to extract the data through the DOM. These extra DOM accesses can be costly operations in what would have otherwise been static data.

Results Our performance evaluation results can be seen in Tables 4.1 and 4.2 for a random set of pages for each application, measured in the Chrome web browser network performance tool. After our experiments finished, we observed that one of the major performance slow downs appeared to be the synchronous loading of script sources. Since our modifications required an increase in the number of external scripts, we modified the pages to use the `async` attribute in the `<script>` tags where possible. This allows those scripts to load asynchronously. This change substantially improved the performance of the applications, but in most cases, not enough to approach the original performance.

Our results show that using CSP for Bugzilla and HotCRP is both a complex task and may harm performance. We show that CSP requires changes to how both applications are structured. While CSP has several desirable properties, such as page level granularity and a fail safe architecture, this shows that, like BEEP and BLUEPRINT, it would be difficult to deploy with a secure setting on complex applications.

4.4 Related Work

There is extensive work how to discover and eliminate XSS vulnerabilities in web applications [72, 138, 82, 83, 140]. There has been work on both eliminating these vulnerabilities on the server and in the client. This work has focused on treating XSS as a bug to be eliminated from an application, keeping XSS vulnerabilities from even reaching production systems. This means that much of this work is static analysis, but some work has focused on dynamic techniques on the server [132]. Other work, specifically KUDZU [105] and FLAX [106], have focused on symbolic execution of client-side JavaScript.

Sanitization, or content filtering, is the elimination of unwanted content from untrusted inputs. Sanitization is applied explicitly by an application or a framework to the untrusted

content. Sanitization is almost always done on the server; generally, the goal is to remove the unwanted content before it reaches the client. Sanitization is usually done as a filter over character elements of string looking for "control" characters, such as brackets in HTML. XSS-GUARD [28] and ScriptGard [107] argue that it is necessary to look at sanitization as a context sensitive problem.

There have also been a number of other HTML security policy systems proposed [84, 95]. We focused on three of the most discussed in the literature, but future evaluations would, of course, need to take these into account as well.

4.5 Towards HTML Security Policies

While current HTML security policy systems are not sufficient for today's web applications because of their performance problems and requirements on how applications are built, they provide very enticing properties. Research should evaluate HTML security policies and how to build better HTML security policy systems. Towards this goal, we start the conversation with several points and questions about HTML security policy systems.

- Researchers should determine the set of properties that an HTML security policy system should have. For example, CSP's page-level granularity is simple as a policy, but puts an undue burden on developers in how they write and retrofit applications. Is this the right trade-off to make?

- From the problems of the systems we observe today, what can we learn? We identified a number of properties systems should not have, such as extensive restrictions on how code is written. Should we just accept these problems to reap the benefit of HTML security policies?

- Combining these systems may be fruitful. For example, a combination approach of BEEP and CSP that allows inlined scripts if they are on a BEEP-like whitelist but also allows external scripts may be an improvement in usability over either system independently. What features can we extract and combine from current systems to build new ones?

- Should new HTML security policy systems work in legacy browsers or focus on state-of-the-art browsers? For example, it seems likely that BLUEPRINTs performance shortcomings could be addressed with better browser support. On the other hand, retrofitting applications for new browser primitives can be a struggle, as seen with CSP.

4.6 Conclusion

HTML security policies should be the central mechanism going forward for preventing content injection attacks. They have the potential to be much more effective than sanitization. However, the HTML security policy systems available today have too many problems to be used in real applications. We presented these issues, including the first empirical evaluation of CSP on real-world applications. New HTML security policy systems and techniques need to be developed for applications to use. As a first step, research needs to identify the properties needed in HTML security policy systems.

Acknowledgments Thanks to my co-authors of this work, Adam Barth and Dawn Song [133], for working with me and giving their permission for this work to appear in this thesis.

Chapter 5

Static Enforcement of Policies for Advertisements

5.1 Introduction

Much of the web economy is fueled by advertising revenue. In a typical advertising scenario, a publisher rents a portion of his or her web page to an advertising network, who, in turn, sublets the advertising space to another advertising network until, eventually, an advertiser purchases the impression. The advertiser then provides content (an advertisement) that the browser displays on the user's screen. Recently, malicious advertisers have sought to exploit these delegated trust relationships to inject malicious advertisements into honest web sites [78, 97]. These attacks are particularly worrying because they undermine confidence in the core of the web economy.

For example, a malicious advertisement exploited The New York Times web site [126] on September 13, 2009. Although most of its advertisements are served via ad networks, The New York Times also includes some advertisements directly from advertisers' servers. In this case, a malicious advertiser posed as Vonage and bought an advertisement. The advertiser then changed the advertisement to take over the entire window and entice users into downloading fake anti-virus software. The New York Times removed the ad but only after a number of visitors were affected.

To protect publishers and end users, advertising networks have been experimenting with various approaches to defending against malicious advertisements. Although there are a wide variety of approaches, ranging from legal remedies to economic incentives, we focus on technological defenses. At its core, displaying third-party advertisements on a publisher's web site is a special case of a mashup. In this chapter, we study whether existing techniques are

This chapter was previously published at the 2010 Network and Distributed System Security Symposium [53].

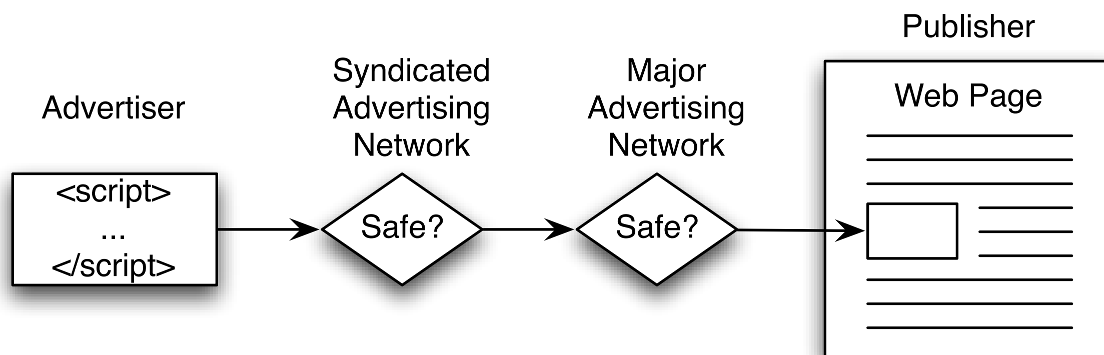

Figure 5.1: A publisher sells space on his or her page to an advertising network. This space may be resold through multiple advertising networks, until it is sold to an advertiser, who provides an advertisement written in a secure JavaScript subset. The advertisement is checked for safety by each advertising network, in turn, and ultimately served to visitors of the publisher's web page.

well-suited for containing advertisements and propose improvements to mashup techniques based on static verifiers.

Static verifiers, such as ADsafe [41], Dojo Secure [145], and Jacaranda [44], are a particularly promising mashup technique for advertising. In this approach, the advertising network (and one or more of its syndicates) verifies that the advertisement's JavaScript source code conforms to a particular subset of the JavaScript language with desirable containment properties (see Figure 5.1). The precise JavaScript subset varies between systems, but the central idea is to restrict the guest advertisement to a well-behaved subset of the language in which the guest can interact only with object references explicitly and intentionally provided to the guest by the host, thus preventing the guest from interfering with the rest of the page.

In this chapter, we focus on evaluating and improving the containment of safe JavaScript subsets that use static verification. Static verifiers are appealing because they provide fine-grained control of the advertisement's privileges. For example, the hosting page can restrict the advertisement to instantiating only fully patched versions of Flash Player, preventing a malicious advertisement from exploiting known vulnerabilities in older versions of the plug-in. However, existing static verifiers do not provide perfect containment. To properly contain advertisements, these systems impose restrictions on the *publisher*: the publisher must avoid using certain JavaScript features that let the advertisement breach containment.

These static verifiers impose restrictions on publishers because of a design decision shared by the existing static verifiers: the verifiers restrict access to object properties using a static blacklist. The blacklist prevents the guest from accessing properties, such as `__proto__`, that can be used to breach containment. The designers of the subsets warrant that their blacklist is sufficient to prevent an advertisement from breaching containment on an otherwise empty page in their supported browsers (or else the subset would always fail to contain advertise-

ments), but the blacklist approach does not restrict access to new properties introduced by the publisher. For example, a publisher might add a `right` method to the string prototype that has not been vetted by the subset designer.

This restriction raises a natural question: how commonly do publishers violate this requirement? If publishers rarely expose new methods (or rarely expose exploitable methods), then this restriction is fairly innocuous. If publishers commonly expose methods that are exploitable, however, then we ought to consider improving these mashup techniques. To answer this question, we designed and implemented an analysis tool that detects which host methods are exposed to guest advertisements. We ran our tool on the non-adult web sites in the Alexa US Top 100 [15] to determine (1) how often sites add methods to prototypes and (2) how many of these methods could be used to breach containment. To measure (1), we rendered the web sites in an instrumented browser that recorded the "points-to" relation among JavaScript objects in the JavaScript heap. Our tool then analyzed the heap and outputted the source code of methods created by the hosting page that would have been exposed to an ADsafe-verified advertisement. To answer (2), we manually analyzed the source code of these methods to determine which sites would have been exploitable by a malicious ADsafe-verified advertisement.

Of the non-adult web sites in the Alexa US Top 100, we found that 59% (53 of 90) exposed new properties to the guest and that 37% (33 of 90) of these sites contained at least one method that could be exploited by an ADsafe-verified advertisement to mount a cross-site scripting (XSS) attack against the publisher. Although the publisher can avoid exposing exploitable methods, even seemingly innocuous methods are exploitable, such as this implementation of `right` found on `people.com`:

```
String.prototype.right = function(n) {
  if (n <= 0) {
    return "";
  } else if (n > String(this).length) {
    return this;
  } else {
    var l = String(this).length;
    return String(this).
        substring(l, l - n);
  }
}
```

We discuss the exploit in detail in Section 5.4. Of the sites with more than 20 exposed properties, only `tagged.com` managed to avoid exposing an exploitable method.

Instead of requiring publishers to vet their exposed methods, we propose eliminating this attack surface by replacing the property blacklist with a whitelist, making static verifiers more robust. Using a technique similar to script accenting [33], we prevent the guest advertisement from accessing properties defined by the hosting page unless those properties are explicitly granted to the guest. Specifically, we restrict the guest to a namespace by requiring all the guest's property names to be prefixed with the guest's page-unique identifier (which

already exists in ADsafe). A guest restricted in this way will not be able to access methods added to the prototype objects by the host because the names of those methods are not in the guest's namespace.

We show that our safe JavaScript subset is as expressive and as easy-to-use as ADsafe by implementing a simple compiler that transforms any ADsafe-verified guest into our proposed subset by prepending the namespace identifier to all property names. Our compiler is *idempotent*: the compiler does not alter code that is already contained in the subset. This property lets us use our compiler as a static verifier for the subset. To check whether a piece of JavaScript code conforms to the subset, one need only run the code through the compiler and check whether the output of the compiler is identical to the input. Idempotency lets each advertising network in the syndication chain apply the compiler to each advertisement without worrying about transforming the advertisement more than once. To protect against a malicious advertising network, the downstream parties (e.g., advertising networks or publishers) can verify each advertisement.

Contributions. We make two main contributions:

- We show that existing static verifiers fall short of defending against malicious advertisements because many publisher web sites use JavaScript features that let the advertisements breach containment.
- We propose a modification to these static verifiers that lets publishers host malicious advertisements without requiring the publishers to rewrite their JavaScript.

Organization. The remainder of this chapter is organized as follows. Section 5.2 surveys related work. Section 5.3 details safe JavaScript subsets based on statically verified containment. Section 5.4 describes our experiment for detecting breaches of containment and reports the results of the experiment. Section 5.5 proposes Blancura, a safe JavaScript subset based on whitelisting. Section 5.6 concludes.

5.2 Related Work

In this section, we survey related work, which falls into two categories: (1) techniques or experiments related to those described herein and (2) other approaches for constructing secure mashups.

Related Techniques and Experiments

In previous work [25], we used our JavaScript heap analysis framework to find browser bugs that cause cross-origin JavaScript capability leaks and compromise the browser's implementation of the same-origin policy. Our framework consists of two pieces: an instrumented browser that records the "points-to" relation among JavaScript objects in the JavaScript

heap and an algorithm for detecting "interesting" edges. In this chapter, we reuse the first piece of the framework in our experiment but replace the second piece with a new algorithm that colors nodes based on whether or not they are accessible to an advertisement running by itself on the empty page. In contrast, our previous algorithm detects pointers leaked from one origin to another, which is not applicable in this setting because all the objects that participate in this sort of mashup are contained in the same origin.

In a series of papers [89, 88, 87], Maffeis, Taly, and Mitchell develop a formal semantics for JavaScript and prove the correctness of a specific static verifier (based on FBJS) in their formal model. In the course of writing their proof, they discovered that the host page can compromise its security by extending the built-in prototype objects. They reported this issue to the designers of ADsafe, who added a restriction on the host's behavior to ADsafe's documentation. However, despite recognizing the issue, the language they propose [88] has the same limitations as other blacklist-based static verifiers. We recommend that they adopt our approach and replace their property-name blacklist with a whitelist.

Finally, Yue and Wang study how often web sites use potentially dangerous JavaScript language features, such as `eval` [142]. Similar to our methodology, the authors render the web pages in question using an instrumented browser. Unlike their study, ours is focused on a specific security issue (extending the built-in prototype objects in a way that lets a malicious advertisement breach containment), whereas their study measures a number of general "security hygiene" properties.

Secure Mashup Designs

Safely displaying third-party advertisements is a special case of the more general *secure mashup problem* in which an integrator (e.g., a publisher) attempts to interact with a collection of untrusted gadgets (e.g., advertisements). The general mashup problem is more difficult than the advertising problem because general mashup techniques aim to provide a high degree of interactivity between the integrator and the gadgets and also among the gadgets themselves.

A number of researchers propose mashup designs based on browser frames (see, for example, [73, 130, 46, 40, 24]). In these designs, the gadget can include arbitrary HTML and JavaScript in a frame. The integrator relies on the browser's security policy to prevent the gadget from interfering in its affairs. In general, these frame-based designs cover a number of use cases, including advertising, but they have a coarse-grained privilege model. For example, HTML5's `sandbox` attribute [8] can restrict the frame's privileges in a number of enumerated ways, but language-based designs can grant finer-grained privileges not preordained by browser vendors. Specifically, a language-based mashup can restrict the gadget to certain (e.g., fully patched) versions of specific plug-ins (e.g., Flash Player), whereas the `sandbox` attribute is limited to allowing or denying all plug-ins.

Gatekeeper [56] is a "mostly static" enforcement mechanism designed to support a rich policy language for complex JavaScript widgets. Although mostly a static analyzer, Gatekeeper uses runtime enforcement to restrict some JavaScript features, such as `eval`. To confine

widgets, Gatekeeper analyzes the whole JavaScript program, including the page hosting the widget. This use of whole program analysis makes Gatekeeper an awkward mechanism for restricting the privileges of advertisements because the advertising network typically does not have the entire source code of every publisher's web site.

Like the static verifiers we study in this chapter, dynamic enforcers, such as Caja [31] and Web Sandbox [4], restrict gadget authors to a subset of the JavaScript language with easier-to-analyze semantics. Unlike static verifiers, dynamic enforcers transform the source program by inserting a number of run-time security checks. For example, Caja adds a dynamic security check to every property access. These dynamic enforcement points give these designs more precise control over the gadget's privileges, typically letting the mashup designer whitelist known-good property names.

Inserting dynamic access checks for every property access has a runtime performance cost. To evaluate the performance overhead of these dynamic access checks, we used the "read" and "write" micro-benchmarks from our previous study [25], which test property access time. We ran the benchmarks natively, translated by each of Caja's two translators (Cajita and Valija), and translated by Web Sandbox. We modified the existing benchmarks in three ways:

1. We modified our use of `setTimeout` between runs to conform to the JavaScript subset under test by using a function instead of a string as the first argument.

2. Instead of running the benchmark in the global scope, we ran the benchmark in a local scope created by a closure. This improved the Valija benchmark enormously because accessing global variables in Valija is extremely expensive.

3. For Cajita and Valija, we reduced the number of iterations from one billion to one million so that the benchmark would finish in a reasonable amount of time. For Microsoft Web Sandbox, we reduced the number of iterations from one billion to 10,000 because the Microsoft Web Sandbox test bed appears to have an execution time limit.

We ran the benchmarks in Firefox 3.5 on Mac OS X using the Caja Testbed [7] and the Microsoft Web Sandbox Testbed [96].

Our findings are summarized in Table 5.1. The Cajita translation, which accepts a smaller subset of JavaScript, slows down the two micro-benchmarks by approximately 20%. Even though Cajita adds an access check to every property access, the translator inlines the access check at every property access, optimizing away an expensive function call at the expense of code size. Valija, which accepts a larger subset of JavaScript, slows down the read and write micro-benchmarks by 1493% and 1000%, respectively. Valija translates every property access into a JavaScript function call that performs the access check, preventing the JavaScript engine from fully optimizing property accesses in its just-in-time compiled native code. Microsoft Web Sandbox performs similarly to Valija because its translator also introduces a function call for every property access. More information about Caja performance is available on the Caja web site [9].

	read	write
Cajita	21%	20%
Valija	1493%	1000%
Microsoft Web Sandbox	1217%	634%

Table 5.1: Slowdown on the "read" and "write" micro-benchmarks, average of 10 runs.

Dynamic enforcers are better suited for the generic mashup problem than for the advertising use case specifically. In particular, advertising networks tend to syndicate a portion of their business to third-party advertising networks. When displaying an advertisement from a syndication partner, an advertising network must verify that the advertisement has been properly transformed, but verifying that a dynamic enforcer's transformation has been performed correctly is quite challenging. By contrast, static verifiers are well-suited to this task because each advertising network in the syndication chain can run the same static verification algorithm.

5.3 Statically Verified Containment

One popular approach to designing a secure JavaScript subset is to *statically verify containment*. This approach, used by ADsafe [41], Dojo Secure [145], and Jacaranda [44], verifies that the contained code is in a well-behaved subset of the JavaScript language. In this section, we describe the characteristics and limitations of the current languages that use this approach.

One important use of these languages is to prevent "guest" code (e.g., advertisements) from interfering with the "host" web page running the guest code. The guest script should be contained by the language to run as if it were run by itself on an otherwise empty page. ADsafe is specifically intended for this use [41], and proponents of Dojo Secure and Jacaranda also envision these languages used for this purpose [45, 145].

Containment Architecture

Secure JavaScript subsets that use statically verified containment prevent guests from using three classes of language features, described below. If left unchecked, a guest could use these language features to escalate its privileges and interfere with the host page. The details of these restrictions vary from language to language. For the precise details, please refer to the specifications of ADsafe, Dojo Secure, and Jacaranda.

- **Global Variables.** These languages prevent the guest script from reading or writing global variables. In particular, these languages require that all variables are declared before they are used (to prevent unbound variables from referring to the global scope) and forbid the guest from obtaining a pointer to the global object (to prevent the guest

from accessing global variables as properties of the global object). For example, these languages ban the `this` keyword, which can refer to the global object in some contexts.

- **Dangerous Properties.** Even without access to global variables, the guest script might be able to interfere with the host page using a number of "special" properties of objects. For example, if the guest script were able to access the `constructor` property of objects, the guest could manipulate the constructors used by the host page. The languages implement this restriction by blacklisting a set of known-dangerous property names.

- **Unverifiable Constructs.** Because dynamically generated scripts cannot be verified statically, these languages also ban language constructs, such as `eval`, that run dynamic script. In addition to dynamic code, the languages also ban dynamic property access via the subscript operator (e.g., `foo[bar]`) because `bar` might contain a dangerous property name at run time. These languages typically do allow dynamic property access via a library call, letting the library check whether the property being accessed at runtime is on the blacklist, and if not, allow the access.

These languages enforce the above-mentioned restrictions using a static *verifier*. The verifier examines the source code of the guest script and checks whether the script adheres to the language's syntactic restrictions. In typical deployments, these languages provide the guest script a library for interacting with the host page. For example, ADsafe provides widgets with a jQuery-style library for accessing a subset of the hosting page's Document Object Model (DOM). These libraries typically interpose themselves between the guest script and the host objects, preventing the guest from interacting with the host in harmful ways. For example, the library might restrict the guest to part of the DOM by blocking access to the `parentNode`. Even in languages that statically verify containment, these libraries often involve dynamic access checks.

Limitations

ADsafe, Dojo Secure, and Jacaranda all block access to dangerous properties using a blacklist. When analyzing a guest script, the static verifier ensures that the script does not use these specific property names. This approach works well on the empty page where the set of property names is known (at least for a fixed set of browsers), but this approach breaks down if the host inadvertently adds dangerous properties to built-in object prototypes because these new properties will not be on the verifier's blacklist [88].

When the host page extends the built-in object prototypes by adding new properties, those properties are visible on all objects from that page. For example, if the host page added the function `right` to `String.prototype`, then the `right` property would be visible on all strings because the JavaScript virtual machine uses the following algorithm to look up the value of a property of an object:

1. Check whether the property exists on the object itself. If so, return the value of the property.

2. Otherwise, continue searching for the property on the object's prototype (identified by the `__proto__` property), returning the value of the property if found.

3. If the object does not have a prototype (e.g., because it's the root of the prototype tree), then return `undefined`.

All the prototype chains in a given page terminate in the `Object.prototype` object. If the host page adds a property to `Object.prototype`, the property will appear on all objects in the page. Similarly, adding a property to `String.prototype` or `Array.prototype` causes the property to appear on all strings or arrays (respectively).

For this reason, ADsafe explicitly admonishes host pages not to extend built-in prototypes in dangerous ways:

> None of the prototypes of the builtin types may be augmented with methods that can breech [sic] ADsafe's containment. [41]

However, ADsafe also says that "ADsafe makes it safe to put guest code (such as third party scripted advertising or widgets) on any web page" [41]. These two statements appear to be in conflict if web sites commonly augment built-in types with methods that breach ADsafe's containment.

5.4 Detecting Containment Breaches

In this section, we evaluate whether existing static verifiers can be used to sandbox advertisements on popular web sites. We automatically detect "unvetted" functions exposed to guest scripts and then examine these function to determine whether they introduce vulnerabilities that could be exploited if the page hosted ADsafe-verified ads.

Approach

To detect when objects created by the host page are accessible to the guest, we load the host page in an instrumented browser. The instrumented browser monitors JavaScript objects as they are created by the JavaScript virtual machine. Our instrumentation also records the "points-to" relation among JavaScript objects. Whenever one object is stored as a property of another object, we record that edge in a graph of the JavaScript heap.

To determine whether an object created by the host page is accessible to the guest, we compare the set of JavaScript objects accessible to the guest script on the empty page with the set accessible to the guest script on the host page. We call an edge from an object in the first set to an object in the second set *suspicious*. The designers of the secure JavaScript subset vet the objects accessible to the guest on the empty page to ensure that guest code

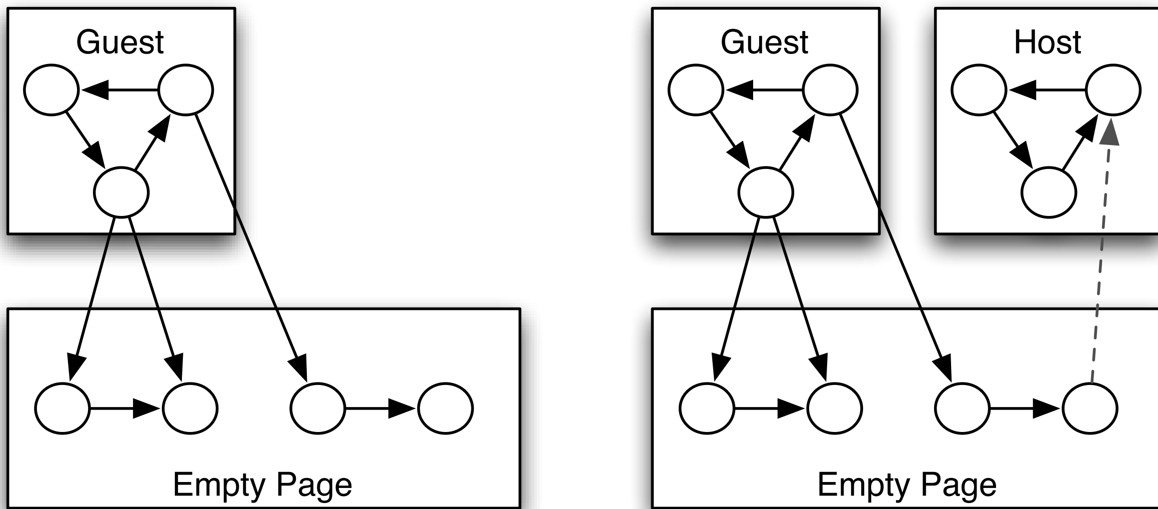

Figure 5.2: Depiction of a simple heap graph. In (a), all nodes are vetted nodes. In (b), a suspicious edge has been detected. The host code has added a pointer that provides the guest code with access to an unvetted object. The suspicious edge points from the prototype of a built-in object to a method defined by the host.

cannot breach containment using those objects (or else the subsets would fail to be secure on virtually every page), but the guest script might still be able to leverage suspicious edges to escape the sandbox.

Not all suspicious edges are exploitable. For example, if the host exposes only the identity function to the guest, the guest cannot leverage this function to escalate its privileges. However, if the host defines a function that calls `eval` on its argument and exposes that function to the guest script, then the guest script can compromise the host completely. To determine whether a suspicious edge lets a guest breach containment, we examined the source code of newly accessible functions. If we are able to construct a proof-of-concept guest script that exploits such a function, we say that the host page has *violated* the containment policy.

Design

To compute the set of suspicious edges, we classify objects in the host page's JavaScript heap into two sets:

- **Vetted.** The *vetted objects* are the JavaScript objects accessible by the guest script on the empty page. In particular, the objects created by the guest script itself are vetted, as are objects reachable from those objects on the empty page. We rely on the designer of the secure JavaScript subset to ensure that these objects are safe for the guest script to access.

- **Unvetted.** The *unvetted objects* are the JavaScript objects that are not accessible by the guest script on the empty page. For example, all the objects created by the host page are unvetted because those objects are not present on the empty page. As another example, the `document` object is unvetted because it is not accessible by the guest script on the empty page, even though `document` exists on the empty page.

We use a *maximal guest* to detect the set of vetted objects. The maximal guest accesses the set of built-in objects allowed by the rules of the safe JavaScript subset.[1] Note that creating multiple instances of these objects does not expand the set of built-in objects reachable by the guest.

We classify the objects in the JavaScript heap for a given page as vetted or unvetted using a two phase algorithm. We load the page in question via a proxy that modifies the page before it is rendered by an instrumented browser.

- **Phase 1:** Before loading the page's HTML, the proxy injects a script tag containing the maximal guest. The instrumented browser seeds the set of vetted objects by marking the objects created by the maximal guest as vetted. We expand the set to its transitive closure by crawling the "points-to" relation among JavaScript objects in the heap and marking each visited object as vetted (with some exceptions noted below). Just prior to completing, the maximal guest calls a custom API to conclude the first phase.

- **Phase 2:** After the maximal guest finishes executing, all the remaining JavaScript objects in the heap are marked as unvetted. As the instrumented browser parses the host page's HTML and runs the host's scripts, the newly created objects are also marked as unvetted. Whenever the instrumented browser detects that the host page is adding an unvetted object as a property of a vetted object, the browser marks the corresponding edge in the heap graph as suspicious and records the subgraph reachable via this suspicious edge. The dashed line in Figure 5.2 represents such a suspicious edge.

One difficulty is knowing when to end the second phase. In principle, a web page could continue to add suspicious edges indefinitely. In practice, however, most of the suspicious edges we found were added by "library code" that is run while the page is loading so that most scripts in the page can make use of the extensions provided by the library. To terminate Phase 2, the proxy inserts a script tag after the page's HTML that calls a custom API.

When computing the set of vetted objects as the transitive closure of the initial vetted objects, the instrumented browser considers only edges in the "points-to" relation that can

[1] For ADsafe, the maximal guest we constructed instantiates one each of the following object types: `Function`, `Object`, `Array`, `RegExp`, `String`, `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. The only built-in objects defined by WebKit but absent from this list are `Number`, `Boolean`, and `Date`. `Date` objects are disallowed in ADsafe code. Numeric and Boolean literals (but not objects) can be constructed by ADsafe code, but these were not included in our maximal guest due to an implementation error. Because of this and other factors, our result regarding the fraction of exploitable sites is a lower bound.

be traversed by guest scripts in the safe subset. For example, in ADsafe, guest scripts cannot access properties with certain names, such as `arguments`, `constructor`, and `eval`. Additionally, ADsafe guests cannot access any properties with names that begin with an underscore character. These properties are ignored in computing the transitive closure, with one notable exception: `__proto__`. Although the guest script cannot access `__proto__` explicitly, the guest script can access the property *implicitly* because this property is used by the algorithm described in Section 5.3 for resolving property names.

Implementation

We implemented our instrumented browser by modifying WebKit, the open-source browser engine that powers Safari and Google Chrome. Our work builds off of our previous JavaScript heap inspection implementation [25]. We implemented our HTTP proxy in Perl using the `HTTP::Proxy` library. The proxy modifies `text/html` HTTP responses in flight to the browser. Because the browser can parse malformed HTML documents with a `<script>` before the main `<html>` and after the main `</html>`, we did not need to modify the response except to prepend and to append the required script blocks.

Experiment

To evaluate whether web sites commonly use JavaScript features that breach the containment of existing statically verified JavaScript subsets, we analyzed the Alexa US Top 100 web sites using our breach detector for ADsafe. This set of web sites was chosen to represent the level of complexity of JavaScript code found on popular web sites. Although this set might not be representative of all web sites, malicious advertisements on these web sites impact a large number of users. We give partial results here. The full results can be found in Appendix B.

Methodology

We retrieved the list of Alexa US Top 100 web sites and tested those web sites in May 2009. Instead of using the home page of every site on the list, we made the following modifications:

- We removed all 9 web sites with adult content. (We were unsure whether our university permits us to browse to those web pages in our workplace.)
- We removed `blogspot.com` because its home page redirected to `blogger.com`, which was also on the list. The `blogspot.com` domain hosts a large number of blogs, but the Alexa list did not indicate which blog was the most popular, and we were unsure how to determine a "representative" blog.
- For some web sites, the home page is merely a "splash page." For example, the `facebook.com` home page is not representative of the sorts of pages on Facebook that contain advertisements. Using our judgment, we replaced such splash pages with more

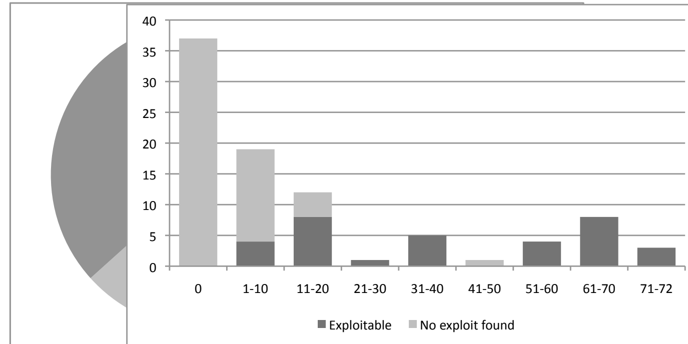

Figure 5.3: Visual depictions of the results of the experiment.

representative pages from the site. For example, on `facebook.com`, we chose the first page presented to the user after login.

We then used the algorithm described above to detect suspicious edges in the JavaScript heap graph of the main frame created by visiting each of the 90 web pages we selected for our experiment. We observed the number of suspicious edges for each page as well as the source code of the exposed methods. We then manually analyzed the source code of the exposed methods to determine whether the suspicious edges would have been exploitable had the selected web pages hosted ADsafe-verified advertisements. Upon finding one exploitable method for a given site, we stopped our manual analysis of the remaining exposed methods for that site because an attacker needs only one vulnerability to construct an exploit.

One potentially fruitful area of future work is to automate this manual analysis step to scale our analysis technique to a larger population of web sites (such as the entire web). In fact, we used a crude static analysis framework based on regular expressions (e.g., `grep` for `return\s+this` and `eval`) to help us find vulnerabilities faster. Even with these simple tools, the manual analysis required only a number of hours.

Results

In our experiment, we observed the following:

- Of the web pages, 59% (53 of 90) contained at least one suspicious edge.
- We were able to construct proof-of-concept exploits for 37% (33 of 90) of the web pages.

Figure 5.3 summarizes our results. We observed a max of 72 suspicious edges and a mean of 16.2 ($n = 90$, $\sigma = 23.8$).

Discussion

Our experiment lower bounds the number of analyzed sites that could be exploitable by a malicious ADsafe-verified advertisement because we might not have found every suspicious

```
String.prototype.right = function(n) {
  if (n <= 0) {
    return "";
  } else if (n > String(this).length) {
    return this;
  } else {
    var l = String(this).length;
    return String(this).substring(l, l - n);
  }
}
```

(a) Relevant host code

```
<div id="GUESTAD_">
<script>
"use strict";
ADSAFE.go("GUESTAD_", function (dom, lib) {
  var f = "Hello".right;
  f(100).setTimeout("... attack code ...", 0);
});
</script>
</div>
```

(b) ADsafe-verified guest code implementing exploit

Figure 5.4: Exploit for `people.com`

edge or every exploit. The histogram in Figure 5.3 shows that the number of suspicious edges is correlated with exploitability. Although not all functions are exploitable (and not all suspicious edges lead to functions), many common JavaScript programming idioms (such as returning `this`) are exploitable. Of the sites with 20 or more suspicious edges, all but one violate containment. The one site that did not violate containment, `tagged.com`, did not contain any unvetted functions.

Case Studies

We illustrate how a malicious guest can use functions supplied by the host page to compromise the security of the sandbox by examining two case studies from popular web sites. In all the cases we examined, when a guest could escalate its privileges using host-supplied functions, the guest could run arbitrary script with the authority of the hosting page, completely breaching containment.

People

We first examine `people.com`, a magazine. People augments the `String` prototype object with a number of utility methods, including a method `right` that extracts the n right-most characters of the string. As an optimization, the function returns the original string (designated by `this`) if it contains fewer than n characters. Although apparently benign, this optimization lets a guest breach containment.

When a JavaScript function is called as a method of an object (e.g., as in `obj.f()`), the `this` keyword refers to that object (e.g., `obj`). However, if a function is called in the global scope (e.g., as in `f()`), then the `this` keyword refers to the global object, commonly known as the `window` object. By removing the `right` function from a primitive string, as shown in Figure 5.4, a malicious guest can call the function in the global scope, binding `this` to the global object. By supplying a sufficiently large value for `n`, the host's function will return the

global object to the attacker. Once the attacker has access to the global object, the attacker can use any number of methods for running arbitrary script with the host's authority.

Twitter

Next, we examine `twitter.com`, a social network. Twitter makes use of the Prototype JavaScript library [10], which adds a number of useful functions to the default JavaScript environment. The exploit we describe for Twitter is applicable to every web site that uses the Prototype Library, including numerous prominent sites [6]. Moreover, we argue that the particular function we exploit in the Prototype Library is not an isolated example because the primary purpose of the Prototype Library is to augment the built-in prototype objects with useful (but potentially exploitable) functions.

The Prototype Library adds a method to the `String` prototype that calls `eval` on the contents of every `<script>` tag found in the string. Exploiting this function is trivial: a malicious advertisement need only create a string containing a `<script>` tag and call its `evalScripts` method (see Figure 5.5). This example illustrates that library designers do not expect their functions to be called with untrusted arguments. This trust assumption is currently valid, as evidenced by the lack of actual exploits against Twitter, but becomes invalid when hosting ADsafe-verified advertisements.

To protect themselves from this attack vector, web sites could remove all potentially exploitable functions, but this approach has two serious drawbacks. First, the publisher must modify its page to the specifications of the advertising network. If an advertising network imposes restrictions on how their publishers code their web sites, publishers are likely to sell their advertising space to another advertising network rather than retrofit their web sites. Second, JavaScript contains many subtly dangerous idioms. Even if a publisher can rid every page of these idioms, the idioms are likely to return as developers modify the site. Instead of forcing each publisher to modify their web pages, we suggest strengthening the sandbox to prevent these prototype modifications from allowing a guest to breach containment.

5.5 Blancura

Current statically verified JavaScript subsets are not robust to prototype extensions. If a web site extends a built-in prototype object with a new method, guest code can access that method and, if that method is exploitable, breach the subset's containment. We propose further restricting guest code to prevent this kind of containment breach. Instead of using a blacklist to ban known-dangerous properties, our system, Blancura, whitelists known-safe properties. In our system, the guest cannot access exploitable methods defined by the host because those functions are not on the whitelist.

```
String.prototype.extractScripts = function() {
  var matchAll = new RegExp(
      Prototype.ScriptFragment, 'img');
  var matchOne = new RegExp(
      Prototype.ScriptFragment, 'im');
  return (this.match(matchAll) || []).
      map(function(scriptTag) {
        return (scriptTag.match(matchOne) ||
                ['', ''])[1];
      });
}
String.prototype.evalScripts = function() {
  return this.extractScripts().map(
      function(script) { return eval(script) });
}
```

(a) Relevant host code

```
<div id="GUESTAD_">
<script>
  "use strict";
  ADSAFE.go("GUESTAD_", function (dom, lib) {
    var expl = '<script language="javascript">' +
        '... attack code ...<\/script>';
    expl.evalScripts();
  });
</script>
</div>
```

(b) ADsafe-verified guest code implementing exploit

Figure 5.5: Exploit for `twitter.com`

Design

To improve isolation, we propose running the host and each guest in separate JavaScript namespaces. By separating the namespaces used by different parties, we can let the host and guest interact with the same objects (e.g., the string prototype object) without interfering with one another. Essentially, our system whitelists access to property names within the proper namespace and blocks access to all other property names. This approach is similar to script accenting [33] and the "untrusted" HTML attribute from [51], both of which separate the namespaces for each distinct principal. For example, we prohibit the guest advertisement from containing the following code:

```
obj.foo = bar;
```

Instead, we require `foo` have a page-unique prefix:

```
obj.BLANCURA_GUEST1_foo = bar;
```

The Blancura verifier enforces this property statically by parsing the guest's JavaScript and requiring that all property names begin with the prefix. For dynamic property accesses (i.e., with the `[]` operator), Blancura enforces namespace separation dynamically using the same mechanism ADsafe uses to blacklist property names.

To generate the page-unique prefix, we leverage the fact that ADsafe guests already contain a page-unique identifier: the widget ID. ADsafe requires that guest JavaScript code appears inside a `<div>` tag with a page-unique `id` attribute, which ADsafe uses to identify the widget. We simply use this value, prefixed by `BLANCURA_`, as the guest's page-unique names-

pace. This technique lets us avoid using randomization to guarantee uniqueness, dodging the thorny problem of verifying statically that a piece of code was "correctly randomized."

Using separate namespaces protects hosts that add vulnerable methods to built-in prototype objects. For example, if the host page adds a vulnerable `right` method to the `String` prototype, a malicious guest would be unable to exploit the method because the guest is unable to access the `right` property. In some sense, this design is analogous to doing an access check on every property access where the property name prefix is the principal making the access request. Ideally, we would modify the host to use a separate namespace (e.g., `BLANCURA_HOST_`), but advertising networks are unable to rewrite the publisher's JavaScript. In practice, however, we do not expect publishers to add properties to the built-in prototypes with guest prefixes, an expectation which is validated by our experiment.

After placing the guest in its own namespace, the guest is unable to access any of the built-in utility methods because those methods have not yet been whitelisted. However, many of those functions are useful and safe to expose to guest script. To expose these methods to the guest, the Blancura runtime adds the appropriate property name:

```
String.prototype.
  BLANCURA_GUEST1_indexOf =
    String.prototype.indexOf;
```

The guest code can then call this function as usual using its prefixed property name, incurring negligible runtime and memory overhead. Instead of blacklisting dangerous built-in methods such as `concat`, this approach lets the subset designer expose carefully vetted methods to the guest individually instead of punting the issue to the developers who use the subset.

Adding prefixes to property names does not measurably affect runtime performance. When the JavaScript compiler parsers a JavaScript program, it transforms property name strings into symbolic values (typically a 32-bit integer), and the symbols generated with and without the prefix are the same. Whitelisting existing properties of a built-in object does incur a tiny memory overhead because more entries are added to the object's symbol table, but this memory overhead amounts to only a handful of bytes per property. Emulating DOM interfaces that use getters and setters (such as `innerHTML`) incurs some run-time cost because the runtime must install a custom getter or setter with the prefixed name. However, systems like ADsafe often wrap the native DOM with a library free of getters and setters [42].

Implementation

We implemented Blancura by modifying the ADsafe verifier. Using a 43 line patch, we replaced the blacklist used by ADsafe's static verifier with a whitelist. By making minimal modifications to the ADsafe verifier, we have a high level of assurance that our implementation is at least as correct as ADsafe's implementation, which has been vetted by experts in a public review process. Additionally, Blancura is a strict language subset of ADsafe, ensuring that any vulnerabilities in Blancura are also vulnerabilities in ADsafe. Our verifier is available publicly at `http://webblaze.cs.berkeley.edu/2010/blancura/`.

To demonstrate that Blancura is as expressive as ADsafe, we implemented a source-to-source compiler that translates ADsafe widgets into Blancura widgets by prefixing each property name with the appropriate namespace identifier. We implemented our compiler by modifying the Blancura verifier to output its parsed representation of the widget. Our compiler is idempotent: if the input program is already in the Blancura subset, the output of the compiler will be identical to the input. The compiler can, therefore, also be used as a Blancura verifier by checking whether the output is identical to the input.

Although we have based our implementation of Blancura on ADsafe, we can apply the same approach to the other JavaScript subsets that use static verification, such as Dojo Secure or Jacaranda. In addition, our approach of whitelisting property names is also applicable to FBJS even though FBJS is based on a source-to-source compiler and not a static verifier. In each case, using the Blancura approach improves the security of the system with minimal cost.

Imperfect Containment

Despite improving the containment offered by static verifiers, a host page can still compromise its security and let a guest advertisement breach containment because JavaScript programs can call some methods implicitly. For example, the `toString` and `valueOf` methods are often called without their names appearing explicitly in the text of the program. If the host page overrides these methods with vulnerable functions, a malicious guest might be able to breach containment. In our experiment, the only site that overrode these methods was Facebook, which replaced the `toString` method of functions with a custom, but non-exploitable, method.

A host page can also compromise its security if it stores a property with a sensitive *name* in a built-in prototype object because guests can observe all the property names using the `for(p in obj)` language construct. Similarly, a guest can detect the presence of other guests (and their widget identifiers) using this technique. We could close this information leak by banning this language construct from Blancura, but we retain it to keep compatibility with ADsafe and because, in our experiment, we did not observe any sites storing sensitive property names in prototype objects.

5.6 Conclusions

Existing secure JavaScript subsets that use statically verified containment blacklist known-dangerous properties. This design works well on the empty page where the set of functions accessible from the built-in prototype objects is known to the subset designer. However, when verified advertisements are incorporated into publisher web sites, the advertisement can see methods added to the built-in prototypes by the publisher. If the publisher does not carefully vet all such functions, a malicious advertisement can use these added capabilities to breach the subset's containment and compromise the publisher's page.

To determine whether this infelicity is a problem in practice, we analyzed the non-adult web sites from the Alexa Top 100 most-visited web sites in the United States. We found that over a third of these web sites contain functions that would be exploitable by an ADsafe-verified advertisement displayed on their site. In fact, we found that once a web site adds a non-trivial number of functions to the built-in prototypes, the site is likely to be exploitable. From these observations, we conclude that JavaScript subsets that use statically verified containment require improvements before they can be deployed widely.

We propose improving these systems by revising a design choice. Instead of blacklisting known-dangerous properties, we suggest whitelisting known-safe properties by restricting the guest advertisement to accessing property names with a unique prefix. This mechanism prevents advertisements from accessing properties installed by the host page while maintaining the performance and deployment advantages of static verification. With our proposal, the web sites in our study do not need to be modified in order to host untrusted advertisements securely.

Acknowledgements. We would like to thank Douglas Crockford, David-Sarah Hopwood, Collin Jackson, Mark Miller, Vern Paxson, Koushik Sen, Dawn Song, David Wagner, and Kris Zyp for feedback and helpful conversations throughout our work on this project. This material is based upon work partially supported by the National Science Foundation under Grant No. 0430585 and the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170.

Additionally, thanks to my co-authors of this work, Matthew Finifter and Adam Barth [53], for working with me and giving their permission for this work to appear in this thesis.

Chapter 6

Translation of JavaScript Towards Verification

6.1 Introduction

JavaScript is the lingua franca of the web. Its highly dynamic nature contributes both to the perception that it is easy to use (at least for small programs), as well as to the difficulty of using it to build secure applications. Whereas traditionally the security threat has been limited by executing JavaScript within the confines of a web browser's sandbox, the scope of JavaScript is now considerably broader, and recent trends indicate that it is likely to broaden further still. For example, Metro[1] applications in the forthcoming Windows 8 operating system can be programmed in JavaScript, and the language can be used to perform a wide range of system calls on the host operating system. Tools that can effectively analyze JavaScript for attacks and vulnerabilities are thus becoming a pressing need.

Reasoning about JavaScript programs, however, poses several challenges. Specifying the semantics of JavaScript is itself a significant research question with which several groups are currently engaged [86, 59, 66]. Rather than address this again, we adopt Guha *et al.*'s solution. Their approach, λJS, gives a translation of a sizable fragment of JavaScript into a Scheme-like dynamically typed lambda calculus. Based on λJS, we developed a tool JS2ML, which translates JavaScript programs to ML. JS2ML, in effect, composes the λJS translation with a Scheme to ML translation, statically typing JavaScript values with a single type representing all dynamic types. What remains then is to reason about the programs emitted by JS2ML[2].

Much of this chapter is under submission and also appears in a technical report [118].

[1] `http://msdn.microsoft.com/en-us/windows/apps/`

[2] While Guha *et al.* demonstrate that their semantics is faithful using an extensive test suite, they make no claims that the λJS translation is fully abstract. One of the novel claims of λJS is that their work is more

Translating JavaScript

We are motivated by prior work that has shown the usefulness of a refined, dependently typed language in verifying security properties for web-related programs [57].

Refinement types allow for the expression of predicates that are assumed to hold across all elements of that type, while dependent types

In that work, the authors require that programmers write their programs in a refined, dependently typed [3], functional language instead of JavaScript. They show that using this, they can verify fined grained policies about the behavior of these programs in relation to the web and web applications. While this work is promising, and clearly shows the value of using type systems to verify security properties of web applications, we look at the problem of automatically translating JavaScript programs to this type of language. Then, we can start looking at the problem of verifying these programs without the developer having to write and reason in a new, complex language. This has the advantage that programmers can write their programs in a familiar, well-known language (JavaScript), and we can do the heavy lifting to set up verification.

There are several distinct challenges in trying to statically detect problems in JavaScript programs through translation. The semantics of JavaScript involves many features and subtle interactions among the features. These include prototype hierarchies, scope objects, implicit parameters, implicit conversions, etc. Following λJS, the JS2ML translation desugars these features into standard ML constructs.

Figure 6.1 illustrates the JS2ML translation—we call programs in the image of the JS2ML translation "MLjs programs". The functions allocObject, select, and update are functions defined in a library for MLjs programs—they serve to create heap-allocated objects and to read and modify their fields. This library also defines a type dyn for dynamically typed values, with constructors like Str, Int, and Fun to inject values into the dyn type.

The definition of the JavaScript function `foo` is translated to the λ-term foo in ML, with two arguments, this, corresponding to the implicit `this` parameter in JavaScript, and an argument args, an object containing all the (variable number of) arguments that a JavaScript function may receive. Just like in JavaScript itself, objects are dictionaries indexed by string-

practical for proofs than prior work, such as [90], but as a result, their semantics is built from an ad hoc understanding of JavaScript rather than a formal translation of the specification. They then automatically test their specification against the extensive Mozilla JavaScript test suite to verify that their semantics matches real JavaScript. While there are no known bugs in the λJS semantics, it cannot be considered a formal semantics because it is verified using a test suite. Thus, since our work is built on λJS, any analysis based on our translation may miss some behaviors of the source program if λJS turns out to be unfaithful to true JavaScript semantics. This limitation of our work could be addressed by using a higher fidelity translation, but we do not see the need for this given the practical power and accuracy λJS demonstrates, and that, in fact, the latest version of λJS includes a mechanized proof of correctness [29].

[3]Dependent types are types that are expressed in relation to a particular value (or values) that an element may hold. Type refinements are types that contain a predicate that are assumed to hold for any element of the given type. For example, in the language F* (which we introduce in Section 6.2), a refinement type might state `x : int { x > 0 }`, which expresses that `x` is always a positive integer. See [116] for more information on these concepts, specifically in the context of F*.

```
1 function foo(x) { this.g = x.f + 1; }
2 foo({f:0});
3 foo = 17;
```

```
1 let foo this args = let x = select args "0" in
2                   update this "g" (plus (select x "f") (Int 1)) in
3 update global "foo" (Fun foo);
4 let args = let x = update (allocObject()) "f" (Int 0) in
5          update (allocObject()) "0" x in
6 apply (select global "foo") global args;
7 update global "foo" (Int 17)
```

Figure 6.1: A JavaScript program (top) and its MLjs version

typed keys, rather than containing statically known field names. In the body of foo, the x argument corresponds to the value stored at the key "0" in the args object. At line 3, the function foo is stored in the global object (an implicit object in JavaScript) at the key "foo". At line 6 we see that a function call proceeds by reading a value out of the global object, calling the apply library function (which checks that its first argument is a Fun v, for some v), and passing two arguments—the global object, the receiver object for the call; and the args object containing a single argument. Finally, at line 7, we update the global object storing the integer 17 at the key "foo".

There are many complications in using this to prove properties about JavaScript. For example, suppose that one wished to prove that the call to the addition function plus at line 2 always returned an integer. One must prove that all callers of foo pass in an object x as the zeroth value in the args object and that x has a field called "f", which contains an integer value. But, even discovering all call-sites of foo is hard—all (non-primitive) function calls occur via lookups into an untyped higher order store (line 6), and this store is subject to strong updates that can change the type of a field at any point (line 7). However, in this work, we focus on the translation specifically. We refer the reader to our report for details on the verification condition generator and how it uses our translation specifically [118].

Contributions

We provide a new method for translating dynamic programs into F⋆ [116], a dependently typed dialect of ML, so that they may then be used in a verifier to statically enforce explicit policies. Our contributions are as follows.

−We develop JS2ML, a translation from JavaScript to ML, and type the resulting (MLjs) programs against the JSPrims API for verification.

−We present JSPrims, a library of dynamic typing primitives in F⋆. This library includes the definition of a type dyn, a new refinement of type dynamic [64].

−We extend JSPrims to include a partial specification of common APIs used by JavaScript programs, including a fragment of the Document Object Model (DOM). Our specification includes properties of recursive data structures (like DOM elements) using inductive predicates.

−We evaluate our work experimentally on a collection of JavaScript web-browser extensions (on the order of 100 lines each). This evaluation uses our JSPrims library and translation to generate well-typed MLjs programs. The use of this translation in a general-purpose verification generator to statically enforce properties is a separate work, and we will provide only a brief description of it here. As mentioned above, for the details on this work, please see our report [118].

Limitations

The main limitation of our work relates to the fragment of JavaScript we currently support. First, because our work is based on λJS, we face similar restrictions as that work. For example, our translation is based on the ECMAScript 3 standard and we consider closed, `eval`-free JavaScript programs, both like λJS. As a further limitation, we also do not currently translate JavaScript exceptions. While this is not a fundamental limitation, it was a choice based on the amount of engineering effort needed in matching the JavaScript semantics of exceptions to the F⋆ model and we did not run into many uses of exceptions in our experiments; we thus leave this for future work. Additionally, while the prototype chain is implicitly present in our translation, it is not explicitly dealt with in the translation or our JSPrims library, which leaves a larger challenge for verification techniques.

Of note, there are many constructs in JavaScript which may appear to be missing from MLjs, but, in fact, are present thanks to the desugaring of λJS. For example, there is no explicit `with` statement equivalent in λJS because it is desugared to a simplified form without an explicit syntax. We highly encourage readers interested in the desugaring of JavaScript to read [60]; in this work, we try to focus on our new contributions, particularly around the typing of JavaScript.

6.2 A brief review of F⋆

F⋆ is a variant of ML with a similar syntax and dynamic semantics but with a more expressive type system. It enables general-purpose programming with recursion and effects; it has libraries for concurrency, networking, cryptography, and interoperability with other .NET languages. In this chapter, aside from the types of ML, we limit ourselves to a few main F⋆-specific typing constructs. Additionally, in this paper, we limit ourselves to a basic description of what the type system of F⋆ provides. For a more thorough description of the language and its design and implementation, please see [115, 116].

The type system of F⋆ is partitioned into several subsystems using a discipline of kinds. We use two base kinds in this chapter. The kind $\star$ is the kind of types given to computations.

In contrast, the kind E is the kind of types that are purely specificational. A sub-kinding relation places the kinds in the order $\star \leq E$.

Ghost refinement types of the form x:t$\{\phi\}$ represent values v:t for which the formula ϕ[v/x] is derivable from the hypothesis in the current typing context. The formula ϕ is itself a type of kind E—it is purely specificational and has no runtime representation. Thus, x:t$\{\phi\}$ is a subtype of t. Dependent function arrows x:t → t' represent functions whose domain is named x of type t and whose co-domain is of type t', where t' may depend on x. For non-dependent function types, we simply write t → t'. We write (x:t ∗ t') for a dependent pair, where x names the first component and is bound in t'. We also use function arrows whose domain are types 'a of kind k and whose co-domain are types t, written 'a::k → t, i.e., these are types polymorphic in types 'a of kind k.

Aside from the base kinds ($\star$ and E), we use product kinds 'a::k ⇒ k' and x:t ⇒ k for type constructors or functions. For example, the list type constructor has kind $\star \Rightarrow \star$, while the predicate Neq (for integer inequality) has kind int ⇒ int ⇒ E. We also write functions from types to types as **fun** ('a::k) ⇒ t, and from terms to types as **fun** (x:t) ⇒ t'. For example, **fun** (x:int) ⇒ Neq x 0 is a predicate asserting that its integer argument x is non-zero. As a general rule, the kind of a type is $\star$, if not explicitly stated otherwise.

F* is also parametric in the logic used to describe program properties in refinement formulas ϕ. These formulas ϕ are themselves types (of kind E, and so purely specificational), but these types can be interpreted in a logic of one's choosing. In this chapter, we use a higher-order logic, extended with a theory of equality over terms and types, a theory of linear integer arithmetic, datatypes, and a select/update theory of functional arrays [94] (useful for modeling heaps). Except for the higher-order constructs (which we handle specially), all decision problems for the refinement logic are handled by Z3, the SMT solver integrated with F*'s typechecker. The binary predicates EqVal and EqTyp below are provided by F*'s standard prelude and represent syntactic equality on values and types respectively. For brevity, we write v=v' and t=t' instead of EqVal v v' and EqTyp t t', respectively.

```
type EqVal ('a::⋆) :: 'a ⇒ 'a ⇒ E
type EqTyp :: E ⇒ E ⇒ E
logic tfun type TypeOf ('a::⋆) :: 'a ⇒ E
```

Every type and data constructor in an F* program is, by default, treated as an injective function in the logic (i.e., according to the theory of datatypes). To override this default behavior, we tag certain declarations with various qualifiers. For example, the declaration of TypeOf above is tagged **logic** tfun, indicating that it is to be interpreted as a non-injective type function from values to E-types in the refinement logic.

Values can also be provided with interpretations in the logic. For example, to model a map type as a functional array in Z3, we write the following declaration in F*.

```
logic array(sel, upd, emp, InDom) type heap
logic val sel : heap → ref 'a → 'a
logic val upd : heap → ref 'a → 'a → heap
logic val emp : heap
logic tfun InDom : heap ⇒ ref 'a ⇒ E
```

This provides an abstract type called heap, with four interpreted functions in the logic: sel selects the value from the heap at the specified reference; upd updates the heap at the specified reference location; emp is the empty heap; and InDom is an interpreted predicate to test whether a reference is allocated in the heap. The **logic val** tag records that these functions are interpreted in the logic only—they may be used only within specifications.

The type system and metatheory of F* has been formalized in Coq, showing substitutivity and subject reduction, while a separate progress proof has been done manually. These type soundness results (modulo the soundness of the refinement logic) imply the absence of failing assertions in any run of a well-typed F* program.

The importance of F* to this work is in the ability to refine the dynamic typing information from our translation. Without the type system of F*, we would be limited in how we can translate JavaScript. Take, for example, a JavaScript variable declaration `var x = 42;` A straightforward ML translation of this would lead to a general, dynamic type for this variable because it can take on any value later in the program and be passed around to functions that take multiple values. For example, given a type dyn that is a sort of union type of all JavaScript types, an ML translation of the JavaScript variable might be **let** (x : dyn) = 42.

However, with type refinement, we might be able to express further conditions on this. In F* we can express a refinement of the above expression that x should be an integer in the expression **let** (x : dyn{EqTyp (TypeOf x) int}), where EqTyp and TypeOf describe type equality and the type of the given variable, respectively. This allows later verification work to create policies and verify them based on the typing information provided by the translation.

The* ML*js language. MLjs is a small subset of F*—its syntax is shown below. Values v include variables, n-ary lambdas, and n-ary data constructor applications. Formally, we encode other common constants (strings, integers, unit, etc.) as nullary data constructors, although our examples use the usual constants. Expressions include values, function application (as in F*, we require the arguments to be values), let bindings, and pattern matching.

Syntax of MLjs

$$v ::= x \mid \lambda \bar{x}.e \mid C\ \bar{v}$$
$$e ::= v \mid v\ \bar{v} \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{match}\ v\ \textbf{with}\ C\ \bar{x} \rightarrow e_1\ \textbf{else}\ e_2$$

No JavaScript-specific constructs appear in MLjs directly. Instead, we provide an F* library, JSPrims, which is used by every MLjs program produced by the JS2ML translation. For example, the data constructors for type dyn are also defined by JSPrims. Additionally, as shown in Figure 6.1, functions to manipulate state are provided by the JSPrims library.

6.3 A library for dynamic typing in F*

As described above, the basic language of our translation, MLjs, is relatively simple in and of itself. We leave much of the complexity in an associated F* library, JSPrims, which

```
type undef :: ⋆
logic type EqTyp :: E ⇒ E ⇒ E
logic type TypeOf ('a::⋆) :: 'a ⇒ E
type dyn =
  | Bool : bool → d:dyn{EqTyp (TypeOf d) bool}
  | Int : int → d:dyn{EqTyp (TypeOf d) int}
  | Str : string → d:dyn{EqTyp (TypeOf d) string}
  | Obj : loc → d:dyn{EqTyp (TypeOf d) object}
  | Undef : d:dyn{EqTyp (TypeOf d) undef}
  | Fun : (dyn → dyn → dyn) → dyn
```

Figure 6.2: A refinement of type dynamic

defines various constructs for expressing properties and later verifying MLjs programs. The translation uses this library to fully express the original JavaScript programs in MLjs.

A refined type dynamic

The first step in creating JSPrims is creating a refinement in F* of the basic dynamic JavaScript type, dyn. Figure 6.2 shows the definition of our type dyn, based on the standard algebraic type dynamic, but with refinement types on each case to recover precision. For example, we have the constructor Int, which allows an integer i to be injected into the dyn type. The type of (Int i) is d:dyn{EqTyp (TypeOf d) int}. (The type of Str is similar.) The refinement formula uses the binary predicate EqTyp, interpreted in the refinement logic as an equivalence relation on types and the type function TypeOf to reference the type of a given variable. Additionally, E is a ghost term in a specification when the proof for the term is irrelevant or available. We refer readers to [118] for an explanation of these terms as they require a more in-depth review of F*.

We use the Obj constructor to promote heap-resident MLjs objects to the dyn type—the type loc, isomorphic to the natural numbers, is the type of heap locations. The Undef constructor is for the `undefined` value in MLjs.

Fun is particularly notable. The JS2ML translation translates every JavaScript function to a 2-ary, curried function. The first argument is for the implicit `this` parameter (made explicit in the translation), and the second parameter is for all the other arguments represented as a dictionary. So, to a first approximation, the type of Fun is (dyn → dyn → dyn) → dyn. In practice, verifiers may modify this type for more precise information. For example, the function's effects on the heap may be worth explicitly modeling, but we leave this to the work of verification. In addition, JSPrims also provides a function application primitive, apply, that unwraps and calls a given Fun.

JSPrims also provides the mechanisms for updating and accessing objects on the heap. The particular model of the heap is left up to later verification work; our default translation simply treats objects as simple mutable arrays. In particular, JSPrims provides allocObject,

```
1 val abstractEquality : dyn → dyn → dyn
2 let abstractEquality (o1 : dyn) (o2 : dyn) = Bool(true)
3
4 val refEquality : dyn → dyn → dyn
5 let refEquality (o1 : dyn) (o2 : dyn) = Bool(true)
6
7 val addition : p1:dyn → p2:dyn → p3:dyn{((IsNumber p1) && (IsNumber p2)) ⇒ IsNumber p3}
8
9 var global : dyn
```

Figure 6.3: Some basic JavaScript primitives in JSPrims

update, and select functions. These provide the functionality for, respectively, allocating a new object, mutating a property in place in an object, and selecting a property from an object.

Our full library includes cases for the other JavaScript primitive types as well. Note, from here on, unless we think they specifically aid understanding, we omit kind annotations on type variables for concision and clarity. We also use the equality operator and write t = t' instead of EqTyp t t'.

One of the most useful portions of the λJS translation is how implicit JavaScript features are desugared and made explicit. For example, the `toString` method is a special JavaScript method that is often called to coerce non-string primitives into strings where strings are semantically required. One case where this is done is in code such as `x[y]`. Here, it is possible that `y` might be an object or a function, or any other non-string JavaScript value. JavaScript implicitly calls `y.toString` dynamically to ensure that the index into `x` is a string. Our translation, following λJS, makes such coercions explicit. Similar ideas apply to functions such as `valueOf` as well. Of note, we implement these implicit coercions as regular JavaScript calls, rather than special cases in the MLjs syntax. Thus, the actual implementations and specifications of these functions is in the JSPrims library.

Modeling JavaScript and the DOM

An important part of JavaScript are many of the objects and functions it provides as a part of the language. In particular, in web applications, JavaScript provides the Document Object Model (DOM) as a programmatic way of accessing the structure of the current page. It is access to this model, the DOM, that JavaScript security policies often wish to restrict [41, 31]. Thus, it is vital that, in our translation, we provide a model of the DOM that is amenable to verification, such as for access control to the DOM or other types of policies.

In the JSPrims library, we provide such a model, both for the DOM and for other core JavaScript functions. Besides being important for verification, it is also vital to the creation of virtually any well-type JavaScript program in MLjs. For example, in Figure 6.3, we see a set of some of the basic primitives in JavaScript that the translation inserts and makes

```
1 val document : dyn{IsObject(document)}
2 val document_body : dyn{IsObject(document_body)}
3 val history : dyn{IsObject(history)}
4 val location : dyn{IsObject(location)}
5
6 val getElementById : p:dyn → q:dyn → dyn
```

Figure 6.4: Some DOM primitives in JSPrims

explicit. Lines 1 and 4 show the definition of abstractEquality and refEquality, respectively[4]. While technically in JavaScript these operators return a dynamic type, we know that, in fact they return a dynamic type that will always carry a boolean value. Thus, this refinement on these values may help verification in understanding JavaScript.

Similarly, on line 7, we see the type definition of addition[5]. This type is particularly important. As with other functions and operators in the translation, it takes arguments with type dyn, and the refinement explicitly points out that these types must have been constructed with one of the number constructors. This, of course, will only type check in F* if, in fact, this is true. If this is proved by the type checker, then a verifier can use this proof further down the line, for example, to show that a variable that was an argument to this function cannot be a string in another context.

Also notable is line 9 where the JavaScript global object is defined. This is a special object in JavaScript that serves as a namespace for all global variables in the program. The global object is accessed (often implicitly) by JavaScript programs to reach many important parts of the program, such as the `document` object in the DOM, which we will discuss shortly.

In Figure 6.4, we see several examples of DOM primitives that are defined in JSPrims. The first four lines define the types for several basic parts of the document: the document itself, the body of the document, the history object, and the location object. These are simple objects but it allows a verification tool to write explicit policies about how these objects are accessed, rather than writing policies abstractly about the DOM.

Take line 6. This defines a basic type for the `document.getElementById` function for accessing elements in the DOM. It is easy to see why a verifier might want to restrict access to this function as it, in turn, effectively gives full access to the DOM of the page. While JSPrims provides this basic object, the translation allows verifiers built on top of it to create a more precise refined type. For example, if you wanted to create an access control policy of what can call `document.getElementById`, you might refine this type as: **val** getElementById : p:dyn → q:dyn{Perm_GetElementById p q} → dyn. Of course, how the Perm_GetElementById refinement works is for the verifier's model to specify.

[4]These correspond to the === and == operators in JavaScript respectively

[5]While this corresponds to JavaScript's + operator, an observant reader might note that, in fact, in JavaScript, + can take many different types as arguments. In the translation, we treat all of these as different functions with an overloaded symbol. Thus, here, we are only looking at the numerical version of addition.

```
type exp =
    | EValue of pos * value
    | EOp1 of pos * prefixOp * exp
    | EOp2 of pos * infixOp * exp * exp
    | EDot of pos * exp * id
    | EBracket of pos * exp * exp
    | ECall of pos * exp * exp option * exp list
    | ECallInvariant of pos * id * id
    | EAssign of pos * id * exp
    | EDotAssign of pos * exp * id * exp
    | EBracketAssign of pos * exp * exp * exp
    | ENew of pos * exp * exp list
    | EObjLiteral of pos * (id * exp) list
    | EList of pos * exp * exp
    | EIf of pos * exp * exp * exp
    | EVar of pos * id * exp * exp

and stmt =
    | SSeq of pos * stmt * stmt
    | SIf of pos * exp * stmt * stmt
    | SWhile of pos * exp * stmt
    | SReturn of pos * exp
    | SBreak of pos
    | SEmpty of pos
    | SExp of exp
    | SForInStmt of pos * id * exp * stmt
    | SVar of pos * id * exp * stmt
    | SVarDecl of pos * id
    | SThrow of pos * exp
    | SComment of string

type prog =
    | PStmt of pos * stmt
```

Figure 6.5: A portion of the JS2ML translation. Note that the pos type represents a program position in the original JavaScript program. Also, for brevity, we elide the definition of value which defines the grammar for JavaScript primitives.

JSPrims provides the basic functionality for the translation to create well-typed MLjs programs. It provides a basic set of types and objects for the core of MLjs to type well-formed JavaScript programs. It leaves great latitude for JavaScript verifiers to build further verification models on top of it.

6.4 Translating JavaScript to ML

The λJS approach to specifying the semantics of JavaScript is attractive. By desugaring JavaScript's non-standard constructs into the familiar constructs of a Scheme-like programming language, λJS makes it possible to apply many kinds of standard program analyses to JavaScript programs. However, after working with the λJS tool, we decided that we were not able to emit the well-typed ML code that we needed with λJS.

We experienced several difficulties with λJS. One example is its relatively brittle front-end. Parsing JavaScript is not easy (mainly because of its unusual semi-colon and off-side rule), and λJS failed to successfully parse many of the programs we were interested in. Additionally, we ran into issues with the translation itself. Given that it was for a simple desugaring, we ran into problems when, for example, we wanted to add in additionally hints regarding type information Thus we decided to implement a tool resembling λJS, JS2ML, from scratch.

Following the main idea of λJS, we implemented JS2ML in a mixture of C# and F#, utilizing a front-end for parsing and desugaring JavaScript provided as part of Gatekeeper [56], a JavaScript analysis framework. Gatekeeper provides many facilities for JavaScript analysis which we have yet to exploit, e.g., it implements a Datalog-based pointer analysis for JavaScript, which we anticipate could be used to generate the hints to our query compiler described previously.

In Figure 6.5, we see the type constructors of the target grammar of MLjs. We have tried to make the labels fairly self explanatory. It consists of expressions and statements, and a program is simply a statement. In the grammar, `pos` is the position in the original JavaScript program that the current expression or statement was derived from, so that we can connect and verification results back to the original program. We elide the definition of JavaScript primitives (integers, strings, regular expressions, etc.) for simplification.

Some of the grammar does require clarification. Building on λJS, many common features of JavaScript are desugared into other constructs. For example, `with` is distilled down to an explicit series of static lookups when referencing variable labels. JavaScript `continue` and `break` statements are desugared to a common `break` statement that jumps to a statically defined label. Regular expression matching is desugared to a function call rather than a separate syntactic form. There are several other constructs similarly desugared, for with we refer you to [60].

Functions, in MLjs, are simply first-class values that are passed around as bindings to variables. Thus, we elide them in Figure 6.5 because they appear in the definition of JavaScript primitives.

Another common point of confusion are the two forms of `if` statements. The standard JavaScript `if-then-else` form is represented by the `SIf` statement structure. However, JavaScript also allows `if` expressions in the form of the ternary operator, `? :`. The MLjs expression `EIf` represents this construct.

In MLjs, the `SVarDecl` and `SVar` statements are for variable declaration and initial variable assignment, respectively. We desugar variable declarations to these two separate statements

because, in fact, they occur at different times. In JavaScript, a variable declaration, even if it appears in the middle of a function, occurs at the beginning of the function. However, any initialization of its value does not actually occur until its appearance in the actual code. We separate the declaration and initialization to make this distinction explicit.

The MLjs expression ECallInvariant is used for manually inserted loop invariants added by the developer or verifier. We discuss the use of this in an upcoming section.

One of the prime benefits of the translation is how it simplifies the structure of programs, making them easier to reason about. For example, note that there are only two types of loops in MLjs, SWhile and SForInStmt. All loops are collapsed into these constructs. It was shown in the prior λJS work that these types of simplifications can make formal reasoning about programs much easier.

Aside from the front-end, and the hints inserted by JS2ML, we describe the principal differences between JS2ML and λJS below.

−The output of JS2ML is typed whereas the output of λJS is untyped. JS2ML inserts the constructors of dyn such as Int and Fun throughout the translation. JS2ML translates select, update, and function application in JavaScript programs to calls of the corresponding JSPrims APIs. In contrast, λJS relies on primitive dynamically-typed operations on references and functions in Scheme.

−JS2ML translates locals to allocated objects on the heap whereas λJS locals are not objects. Our object-based locals makes our heap model more uniform.

−JS2ML has a simple annotation language for stating loop invariants. However, the support for annotations is very primitive at present—we plan to improve on this in the future.

−Guha *et al.* check the accuracy of their tool against a large corpus of benchmarks, showing the output of programs under the λJS semantics was identical to the original JavaScript. We have not yet tested JS2ML at a similar scale, although we plan to in the future.

Towards Verification The purpose of the translation is to be used by a verifier to prove properties of JavaScript programs. We do not claim work on verification as part of the results of this work. For full details on how verification can be done, we refer you to [118]. Here we describe some of the features of our translation that relate to verification.

The verification in [118] relies on a general-purpose verification generator designed for ML programs. The translation generates JS2ML programs that use the JSPrims library to verify against. The translation leads to basic, well-typed programs, with several other features for the verifier:

−For part of the proofs by the verifier, it requires hand coded loop invariants. Thus, the translation adds an ability for a developer to, in the comments, provide a loop invariant. This allows the developer to explicitly set the loop invariants which the verifier might not otherwise be able to automatically extract. This can be seen in the type constructor in Figure 6.5 line 8.

−One of the real keys to the verification process is the set of assertions set in the JSPrims library. The library provides the basic set of functions that define the JavaScript libraries,

the DOM, and other basic functions of JavaScript. This is what the translated programs are type checked against. This is also where the set of assertions can be set that provide the properties the program will be verified against. That is, the JSPrims library provides assertions about properties of the program which the verifier then attempts to prove via the type system. By default, this is simple program correctness conditions. However, more complicated correctness or security properties can be defined here.

The JS2ML translation is, at its most basic level, a well-typed version of JavaScript in ML. However, the addition of this type information as well as the simplification of the structure of the JavaScript language help the verifier produce proof obligations. The actual verification of these programs is not part of this work.

6.5 Examples

This section presents several examples of our translation. All the major web browsers provide support for JavaScript extensions, which can provide a range of features for an enhanced browsing experience. Browser extensions have been the subject of some recent study, both because of their popularity, and because of the security and reliability concerns they raise [57, 21, 23]. Extensions are executed on every page a user visits, and runtime errors caused by extensions can compromise the integrity of the entire browser platform. Thus, a methodology to verify security policies on extensions is of considerable practical value, so we chose these as a first step in our evaluation of our translator.

For our evaluation, we looked dozens of microbenchmarks, but more importantly, 11 real Chrome extensions [23]. The translation translates each of these programs in a matter of a few seconds, so we do not discuss the performance here. In this section, we review two browser extensions for the Google Chrome web browser, and a look at parts of our translation[6]. These extensions are based on extensions that were studied in prior work [57].

Example 1: HoverMagnifier

Our first extension is HoverMagnifier, an accessibility extension: it magnifies the text under the cursor, presumably to assist a visually impaired user. The core interface of HoverMagnifier with a web page is implemented in 27 lines of JavaScript. A key part of its code is shown below — it involves the manipulation of a collection of DOM elements. At line 1 it calls the DOM function `getElementsByTagName` to get all the `<body>` elements in a web page. Then, line 3 sets an event handler, `magnify`, to be called whenever the user's mouse moves—we elide the definition of `magnify`.

[6] Also note that we have "cleaned up" these translations to make them more human readable. For example, in **let** bindings, the translation would normally generate randomized variable names, but we have modified these to create more readable names.

A fragment of HoverMagnifier

```
1 var body = document.getElementsByTagName('body')[0];
2 function magnify(evt) { ... }
3 body.onmousemove = function (evt) { magnify(evt); };
```

We have to provide specifications for all the APIs used by the program, which we do in JSPrims. For our extensions, this API is principally the DOM. We have already seen a small fragment of our DOM specification. We also need basic functions of JavaScript that are normally implicit, such as property access and function application, explicit.

We see the translation of the `body.onmousemove` assignment in the display below. There are several important points here. For example, on line 3, we see an explicit object allocation. This creates a heap location for a new JavaScript object, which is normally an implicit process. This also allows for refinements that specify that a variable is assigned an object exclusively.

Additionally, on line 5, we see the call to magnify inside the function definition. This requires an selection from the global object to get the `magnify` function, but also a special call to apply to finally make the call. This is needed because function objects are also of type dyn, and thus need to be "unwrapped" to call the function.

This particular example also shows an example of where the translation might help a verifier. Imagine a policy that restricts access to event callbacks, such as `onmousemove`. The translation provides the type information on the global object such that the verifier can statically show the global object to be in the first select statement. Then the verifier is obligated to to prove or disprove that the properties accessed via the select and update lead to `onmousemove`. How a verifier might do this is shown in [118].

Translation of basic magnify event in HoverMagnifier

```
1 let _ = update (select (Obj global) "body") "onmousemove"
2         (Fun ((fun (this:dyn) → (fun (args:dyn) →
3                 locals = allocObject () in
4                 let _ = update locals"evt" (select args "0") in
5                 let _ = apply (select (Obj global) "magnify")(Obj global)
6                         (let args = allocObject() in
7                         let _ = update args "0" (select local "evt") in args) in
8                                 Undef)))) in
```

Example 2: Typograf

Our second example is Typograf, an extension that formats text a user enters in a web form. We show a small (simplified) fragment of its code below.

Message passing with callbacks in Typograf

```
1 function captureText(elt, callback) {
2  if (elt.tagName === 'INPUT'){ callback({ text: elt.value }); }
3 }
4 function listener(request, callback) {
5  if (request.command === 'captureText') {
6   captureText(document.activeElement, callback);
7 }}
8 chromeExtensionOnRequest.addListener(listener);
```

Typograf, like most Chrome extensions, is split into two parts, content scripts which have direct access to a web page, and extension cores which access other resources. The two parts communicate through message passing. When Typograf's content script receives a request from the extension core to capture text, it calls captureText, which calls the callback function in the request (line 2). At line 8, Typograf registers listener as an event handler with the Chrome extension framework, by calling the function addListener. Below is the translation for this line.

Translation of Typograf Chrome API call

```
1 let _ = apply (select (select (Obj global) "chromeExtensionOnRequest")
2                                "addListener")
3                         (select (Obj global) "chromeExtensionOnRequest")
4                         listener in ...
```

The translation here is relatively straightforward, with a series of select calls to access the *chromeExtensionOnRequest* object. This shows how the simplification can help a verifier. While the simple type of the chromeExtensionOnRequest object (which is defined in JSPrims is of dyn, a policy may want to restrict access to this. Thus, a verifier might refine its type to something like chromeExtensiononRequest:dyn{RequiresRequestPerm(chromeExtensionRequest)}, where RequiresRequestPerm is a predicate that defines a permission that the extension must be granted for it to resolve to true. Then, if select is refined to require objects of type dyn but with valid permissions, the verifier will have to prove that the caller has permission to access this function before proceeding.

Another part of the Typograf translation is shown below. It contains many similar features to HoverMagnify. Like HoverMagnify, it emphasizes the explicit description of the heap in the translation. While relatively straightforward for the translation, this has important implications for verifiers as this is important for their modeling.

Partial translation of Typograf

```
1 let eq = refEqBool (select (select locals "request") "command") (Str "captureText") in
2 if eq then
3         let _ = apply (select (Obj global) "captureText") (Obj global)
4                 (let args = allocObject () in
5                 let _ = update args "0" (select (select (Obj global) "document") "activeElement") in
6                 let _ = update args "1" (select locals "callback") in
7                         args) in
8         Undef
9 else ...
```

6.6 Using the Translation for Verifying Correctness and Security

Ultimately, the goal of this translation is to provide a mechanism on which to built verifiers to check security policies on JavaScript programs. Previous work has shown the value of type checking for verifying security policies of web applications (and, in particular, browser extensions) [57]. In this section, we briefly describe how our translation relates to such work, and what the results of tools built on our tool have been.

Our translation takes JavaScript programs and outputs a well-typed sub-language of F*, MLjs. We provide a library, JSPrims, that describes the DOM as well as a great deal of the standard functionality of standard JavaScript, so that programs will be well-typed, but also as a starting block for doing complex verification of JavaScript programs.

Our translation puts in place the basic tools for building a verifier of security properties in JavaScript by generating well-typed MLjs and creating a model of the DOM whose type system can be used to verify properties. For example, we have described how one might create an access control policy using the type system to prevent a program from accessing parts of the Chrome extension API. Such a verifier could by used to check properties of JavaScript programs. This would provide guarantees about the behavior *of the JavaScript itself* (assuming soundness and correctness of our translation), so the original JavaScript can be executed and assumed to have the desired properties. While, in fact, MLjs is a fully executable language on top of the .NET framework, we do not particularly encourage its use as an independent executable language or view it with that goal in mind; our hope is to provide proofs about the original JavaScript which can then be executed as-is.

In fact, in [118], the authors build a verifier on top of our translation. This chapter shows that JavaScript programs can be verified using a general-purpose verification tool, built on top of our translation. There are several challenges that the paper addresses, such as how to model the heap using our translation, and some of the difficulties in modeling stateful expressions, which the authors use the Hoare state monad to address. They obtain higher-order verification conditions for JavaScript programs that can be discharged by an off-the-shelf automated theorem prover, specifically Z3 [47]. The authors use and extend

jsprims to enforce specific properties. This provides a fully mechanized proof of soundness, by virtue of the soundness of F*. In their experiments, they apply the tool chain to the browser extensions shown above to show the absence of runtime errors.

While we only deal with the actual translation of JavaScript in this work, [118] shows a very important use of our translation in verifying JavaScript. We believe that translation can be used in the future to verify more complex security properties of JavaScript as well. While at first glance, the dynamic nature of JavaScript appears to make it not amendable to traditional verification work, we believe that translating JavaScript will allow us to apply more and more well known techniques from verification to ensure security properties, thus getting the dynamic benefits of JavaScript while maintaining many of benefits of a more statically typed language.

6.7 Related Work

One of the inspirations for this work is work by Guha, et al. on verifying security policies of web browser extensions [57]. In this work, the authors build a system for writing browser extensions in Fine (the predecessor to F*) and using the type system to statically verify the security properties. While their approach is very successful, instead of requiring authors to write extensions in a functional style with complex, unfamiliar types, we approach the problem by taking off-the-shelf extensions written in JavaScript and seeing if we can translate programs to a more complex type system automatically.

There is a long tradition of aiming to equip dynamic languages with a notion of static typing, starting perhaps with [64]. Following that line of work, [65] defined a translation from Scheme to ML by encoding Scheme terms using the algebraic ML type dynamic. Our work is related to this line in two regards. First, our JS2ML translation makes use of a similar Scheme to ML translation (combined with λJS, which we have already discussed). Second, Henglein and Rehof were also able to statically discover certain kinds of runtime errors in Scheme programs via their translation to ML. Our methodology also aims to statically detect errors in dynamically typed programs via a translation into a statically typed language. Of course, because of the richness of our target language, we are able to verify programs in a much more precise (and only semi-automated) manner. Besides, we need not stop at simply proving runtime safety—our methodology enables proofs of functional correctness.

There are many systems for inferring or checking types for dynamic languages—too many that covering all of them thoroughly is impossible. We focus primarily on the prior works that rely on a mechanism similar to ours, i.e., approaches based on dependent typing. Dminor [27] provides semantic subtyping for a first-order dynamically typed language. [122] provide refinement types for a pure subset of Scheme. System D [36] is a refinement type system for a pure higher-order language with dictionary-based objects. None of these systems handle both higher-order functions and mutable state. Our translation explicitly has these features in mind for building verifiers for JavaScript. Additionally, we show how to embed a dynamically typed programming language within a general-purpose dependently

typed programming language. This has several benefits. In contrast to the prior work, each of which required a tricky, custom soundness proof (and in the case of System D, even a new proof technique), our approach conveniently rides on the mechanized soundness result for F*. Furthermore, implementing a new type system or program verifier is a lot of work. We simply reuse the implementation of F*, with little or no modification. This has the benefit that verifiers use our translation, they can leverage tools that already exist for F* for free.

In other work, [54] show how to write specifications and reason about the DOM using context logic. Our specification of the DOM, in contrast, uses classical logic, and is not nearly as amenable to modular reasoning about the DOM, which has many complex aliasing patterns layered on top of a basic n-ary tree data structure. Understanding how to better structure our specifications of the DOM, perhaps based on the insights in [55], is another line of future work.

Many tools for automated analyses of various JavaScript subsets have also been constructed. Notable among these are two control-flow analyses. We have already mentioned Gatekeeper, a pointer analysis for JavaScript—our JS2ML implementation shares infrastructure with this tool. The CFA2 analysis [127] has been implemented in the Doctor JS tool to recover information about the call structure of a JavaScript program. Our method of reasoning about JavaScript programs by extracting heap models in Z3 can also be seen as a very precise control flow analysis. However, as we have already discussed, there is ample opportunity for our tool to be improved by consuming the results of a source-level control-flow analysis as hints to our solver.

6.8 Conclusions

JavaScript has a dubious reputation in the programming language community. It is extremely popular, but is also considered to have a semantics so unwieldy that sound, automated analysis is considered an extremely hard problem. Previous work has begun to question this, however, by rewriting JavaScript to make the semantics more explicit.

In this work, we expand on this idea by translating JavaScript into a statically, refined, and dependently typed language, MLjs, which is a subset of F*. We show how JavaScript can be translated into MLjs in a well-typed way. As part of this process, we build a library, JSPrims to model standard portions of JavaScript as well as the DOM. Furthermore, we suggest how our translation might be used to verify security properties of JavaScript programs. In fact, our tool has already been used to build a more complex verification tool on top of our translation.

Type checking for security policies of web applications has been shown to be very promising. In this work, we have shown a vital first step in building verifiers for this purpose. Our tool serves as the basis for verification tools by providing a method of generating dependent, refined, well-typed JavaScript alongside a model of JavaScript and the DOM.

Note For the latest information on this project, including the code associated with this work, please visit: `http://research.microsoft.com/en-us/projects/fstar/`. At the time of the publication of this thesis, the work associated with this chapter can primarily be found in the `js2fs` and `examples/monadic/js` directories in the F* 0.6-alpha download.

Acknowledgments Thanks to my co-authors of this work, Nikhil Swamy, Juan Chen, Ben Livshits, and Cole Schlesinger [118], for working with me and giving their permission for this work to appear in this thesis.

Chapter 7

Conclusion

The web has become increasingly sophisticated, and with it, so have web applications. These applications use untrusted data in many new ways and understanding the policies that developers use to secure untrusted data is vitally important. We must to understand the tools available to developers and to constantly strive towards new tools that meet their needs. In this thesis, we showed several examples of how to achieve these goals.

First, in Chapter 3, we measured the current state of tools in web frameworks in expressing implicit security policies on untrusted data. We built a model of web browsers to show the complexity that any web application must deal with in using untrusted data in its content. We compared the tools that web frameworks grant and the policies they can enforce to the behavior of real applications and showed that there is a substantial gap.

Next, in Chapter 4, we examined and measured a current explicit policy system, Content Security Policy (CSP). We examined the many benefits of Content Security Policy and compared it to other techniques and technologies, showing its many advantages. We also quantified and measured some of the problems and difficulties it presents, showing that it has many potentially unforeseen pitfalls. We then suggested several possible paths toward solving these problems.

Then, in Chapter 5, we looked at a current policy mechanism for embedding advertisements, ADsafe. We built a system for dynamically checking the policy that it purports to enforce. By running a set of experiments on real-world websites, we showed that, in fact, it fails to enforce its own policies. We then showed how to fix these problems by introducing a new static verifier, Blancura, that addresses these issues.

Finally, in Chapter 6, we looked towards more powerful, future techniques in policy enforcement through type checking. Motivated by prior work that showed great promise for statically enforcing security policies on web applications via type checking, we built a translation from JavaScript programs into a well-typed language, MLjs. Utilizing many of the important features of its parent language, $F^{\star}$, we show that MLjs provides an important step towards the automated verification of JavaScript.

We believe that it's vitally important to scientifically measure and understand how systems implicitly apply security policies to untrusted data in web applications. If we do not,

we risk having an unfounded belief in the security of our systems, or even worse, no way of protecting our applications. We believe that there is much future work to be done in measuring and building new security technologies for web applications, but this work has provided an important first step.

In this thesis, we have shown that we can understand and improve implicit web security policies. We can study systems in place today to quantitatively and qualitatively understand the advantages and problems of how they express security policies for untrusted data. Furthermore, we have shown several novel techniques for improving on and building new systems for enforcing security policies on web applications.

Appendix A

Transductions in the Browser

Table A.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 A.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 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 3.1. 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 negated.

One set of DOM read access APIs creates a serialized string of the AST representation of an element, as described in Table A.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 A.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 3.1 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).

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 A.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 3.1.

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¿>¡/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¿>¡/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 A.2: Details regarding the transducers mentioned in Table A.1. They all involve various parsers and serializers present in the browser for HTML and its related sub-grammars.

Appendix B

Alexa US Top 100

We list here the subset of the sites in the Alexa US Top 100 sites that we used in our experiment. Note that the subset does not include the nine adult sites listed at the time of access, nor does it include `blogger.com` because it is the same site as `blogspot.com`. Alexa was accessed on May 1, 2009 to obtain the list of sites.

Rank	**Website**	**# Suspicious Edges**
1	google.com	0
2	yahoo.com	0
3	facebook.com	17
4	youtube.com	2
5	myspace.com	36
6	msn.com	14
7	live.com	0
8	wikipedia.org	0
9	craigslist.org	0
10	ebay.com	13
11	aol.com	9
12	blogspot.com	13
13	amazon.com	2
14	go.com	0
15	cnn.com	33
16	twitter.com	60
17	microsoft.com	35
18	flickr.com	12
19	espn.go.com	1
20	photobucket.com	60

Rank	Website	# Suspicious Edges
21	wordpress.com	0
22	comcast.net	62
23	weather.com	0
24	imdb.com	0
25	nytimes.com	61
26	about.com	0
27	doubleclick.com	1
29	linkedin.com	0
30	apple.com	16
31	cnet.com	72
32	verizon.net	0
33	vmn.net	18
34	netflix.com	17
35	hulu.com	63
36	mapquest.com	5
37	att.net	0
38	rr.com	70
39	adobe.com	59
40	foxnews.com	52
42	ask.com	0
43	mlb.com	3
44	rapidshare.com	0
45	answers.com	3
46	walmart.com	0
48	fastclick.com	0
51	reference.com	10
52	bbc.co.uk	0
53	target.com	0
54	tagged.com	44
55	ning.com	0
56	careerbuilder.com	37
57	dell.com	0
59	disney.go.com	0
60	digg.com	1

Rank	Website	# Suspicious Edges
61	att.com	0
62	usps.com	63
63	typepad.com	0
64	wsj.com	64
65	ezinearticles.com	0
66	bestbuy.com	1
67	foxsports.com	0
68	livejournal.com	20
69	thepiratebay.org	14
70	ehow.com	13
71	imageshack.us	2
72	tribalfusion.com	0
74	aweber.com	0
75	ups.com	0
76	megavideo.com	0
77	yelp.com	66
78	mozilla.com	0
79	deviantart.com	9
80	expedia.com	0
82	pandora.com	0
83	nba.com	61
84	newegg.com	28
86	irs.gov	0
87	washingtonpost.com	0
88	cox.net	0
89	dailymotion.com	72
91	download.com	72
92	reuters.com	2
93	zedo.com	2
94	monster.com	17
95	people.com	1
96	verizonwireless.com	37
97	realtor.com	1
98	ign.com	9
99	pogo.com	0
100	latimes.com	1

Table B.1: Subset of the Alexa US Top 100 sites used in the experiment.

Bibliography

[1] Bugzilla. http://www.bugzilla,org/.

[2] HotCRP. http://www.cs.ucla.edu/~kohler/hotcrp/index.html/.

[3] jQuery. http://jquery.com/.

[4] Live Labs Web Sandbox. http://websandbox.livelabs.com/.

[5] OWASP: Top 10 2007. http://www.owasp.org/index.php/Top_10_2007.

[6] Who's using Prototype. http://www.prototypejs.org/real-world.

[7] Caja Test Bed, August 2009. http://cajadores.com/demos/testbed/.

[8] HTML 5: A vocabulary and associated APIs for HTML and XHTML, April 2009. http://www.w3.org/TR/2009/WD-html5-20090423/.

[9] Performance of cajoled code, August 2009. http://code.google.com/p/google-caja/wiki/Performance.

[10] Prototype JavaScript Framework, May 2009. http://www.prototypejs.org/.

[11] Content security policy (csp), 2012. http://developer.chrome.com/extensions/contentSecurityPolicy.html.

[12] Introducing content security policy, 2012. https://developer.mozilla.org/en-US/docs/Security/CSP/Introducing_Content_Security_Policy.

[13] An introduction to content security policy, 2012. http://www.html5rocks.com/en/tutorials/security/content-security-policy/.

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

[15] alexa.com. Alexa top 500 sites. http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none, 2008.

[16] How To: Prevent Cross-Site Scripting in ASP.NET. `http://msdn.microsoft.com/en-us/library/ff649310.aspx`.

[17] Microsoft ASP.NET: Request Validation – Preventing Script Attacks. `http://www.asp.net/LEARN/whitepapers/request-validation`.

[18] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, E.P. Markatos, and T. Karagiannis. xJS: practical XSS prevention for web application development. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 13–13. USENIX Association, 2010.

[19] Elias Athanasopoulos, Vasilis Pappas, and Evangelos Markatos. Code injection attacks in browsers supporting policies. In *Proceedings of Web 2.0 Security and Privacy 2009*, 2009.

[20] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[21] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities, 2010.

[22] 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*, pages 360–371, Washington, DC, USA, 2009. IEEE Computer Society.

[23] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities, 2010.

[24] Adam Barth, Collin Jackson, and William Li. Attacks on JavaScript Mashup Communication. In *Web 2.0 Security and Privacy Workshop 2009 (W2SP 2009)*, May 2009.

[25] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proc. of the 18th USENIX Security Symposium (USENIX Security 2009)*. USENIX Association, August 2009.

[26] 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, pages 91–100, New York, NY, USA, 2010. ACM.

[27] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an smt solver. In *ICFP '10*. ACM, 2010.

[28] Prithvi Bisht and V. N. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 23–43, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Brown PLT. Mechanized lambdajs. `http://brownplt.github.com/2012/06/04/lambdajs-coq.html`, 2012.

[30] Frederic Buclin. Bugzilla usage world wide. `http://lpsolit.wordpress.com/bugzilla-usage-worldwide/`.

[31] google-caja a source-to-source translator for securing javascript-based web content. `http://code.google.com/p/google-caja/`.

[32] CakePHP: Sanitize Class Info. `http://api.cakephp.org/class/sanitize`.

[33] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 2–11, New York, NY, USA, 2007. ACM.

[34] Erika Chin and David Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM workshop on Secure web services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[35] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[36] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In *POPL '12*, 2012.

[37] ClearSilver: Template Filters. `http://www.clearsilver.net/docs/man_filters.hdf`.

[38] CodeIgniter/system/libraries/Security.php. `http://bitbucket.org/ellislab/codeigniter/src/tip/system/libraries/Security.php`.

[39] CodeIgniter User Guide Version 1.7.2: Input Class. `http://codeigniter.com/user_guide/libraries/input.html`.

[40] Steven Crites, Francis Hsu, and Hao Chen. OMash: enabling secure web mashups via object abstractions. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 99–108, New York, NY, USA. ACM.

[41] Douglas Crockford. Adsafe : Making javascript safe for advertising. `http://www.adsafe.org/`.

[42] Douglas Crockford. ADsafe DOM API. `http://www.adsafe.org/dom.html`.

[43] Ctemplate: Guide to Using Auto Escape. `http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html`.

[44] David-Sarah Hopwood. Jacaranda Language Specification, draft 0.3. `http://www.jacaranda.org/jacaranda-spec-0.3.txt`.

[45] David-Sarah Hopwood. Personal Communication, June 2009.

[46] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: secure component model for cross-domain mashups on unmodified browsers. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 535–544, New York, NY, USA. ACM.

[47] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[48] django: Built-in template tags and filters. `http://docs.djangoproject.com/en/dev/ref/templates/builtins`.

[49] Django sites : Websites powered by django. `http://www.djangosites.org/`.

[50] The Django Book: Security. `http://www.djangobook.com/en/2.0/chapter20/`.

[51] Adrienne Felt, Pieter Hooimeijer, David Evans, and Westley Weimer. Talking to strangers without taking their candy: isolating proxied content. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 25–30, New York, NY, USA, 2008. ACM.

[52] Matthew Finifter and David Wagner. Exploring the Relationship Between Web Application Development Tools and Security. In *Proceedings of the 2nd USENIX Conference on Web Application Development*. USENIX, June 2011.

[53] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing capability leaks in secure javascript subsets. In *Proc. of Network and Distributed System Security Symposium, 2010*.

[54] Philippa A. Gardner, Gareth D. Smith, Mark J. Wheelhouse, and Uri D. Zarfaty. Local hoare reasoning about dom. In *PODS '08*. ACM, 2008.

[55] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for javascript. In *POPL '12*. ACM, 2012.

[56] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proc. of the 18th USENIX Security Symposium (USENIX Security 2009)*. USENIX Association, August 2009.

[57] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Proceeding of the IEEE Symposium on Security and Privacy, 2011*, 2011.

[58] 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, pages 561–570, New York, NY, USA, 2009. ACM.

[59] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *ECOOP'10*. Springer-Verlag, 2010.

[60] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.

[61] Matthew Van Gundy and Hao Chen. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. *16th Annual Network & Distributed System Security Symposium*, 2009.

[62] Google Web Toolkit: Developer's Guide – SafeHtml. `http://code.google.com/webtoolkit/doc/latest/DevGuideSecuritySafeHtml.html`.

[63] R. Hansen. XSS cheat sheet, 2008.

[64] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.

[65] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for scheme: Translating scheme to ml. In *FPCA*, pages 192–203, 1995.

[66] David Herman and Cormac Flanagan. Status report: specifying javascript with ml. In *Proceedings of the 2007 workshop on Workshop on ML*, ML '07, pages 47–52. ACM, 2007.

[67] Ian Hickson. HTML 5 : A vocabulary and associated apis for html and xhtml. `http://www.w3.org/TR/html5/`.

[68] 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)*, pages 188–198, June 2009.

[69] HTML Purifier Team. Css quoting full disclosure, 2010. `http://htmlpurifier.org/security/2010/css-quoting`.

[70] HTML Purifier : Standards-Compliant HTML Filtering. `http://htmlpurifier.org/`.

[71] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security*, 2010.

[72] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[73] Collin Jackson and Helen J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 611–620, New York, NY, USA, 2007. ACM.

[74] John Jean. Facebook CSRF and XSS vulnerabilities: Destructive worms on a social network. `http://seclists.org/fulldisclosure/2010/Oct/35`.

[75] JiftyManual. `http://jifty.org/view/JiftyManual`.

[76] T Jim, N Swamy, and M Hicks. Beep: Browser-enforced embedded policies. *16th International World World Web Conference*, 2007.

[77] 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.

[78] Dan Kaplan. Malicious banner ads hit major websites, September 2007. `http://www.scmagazineus.com/Malicious-banner-ads-hit-major-websites/article/35605/`.

[79] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis*, 2009.

[80] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM, 2006.

[81] KSES Developer Team. Kses php html/xhtml filter.

[82] Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, 2005.

[83] Benjamin Livshits, Michael Martin, and Monica S. Lam. SecuriFly: Runtime protection and recovery from Web application vulnerabilities. Technical report, Stanford University, September 2006.

[84] Benjamin Livshits and Úlfar Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, 2007.

[85] M.T. Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 331–346. IEEE, 2009.

[86] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS'08*, volume 5356 of *LNCS*, 2008.

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

[88] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[89] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[90] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, 2008.

[91] 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*, pages 125–140, Washington, DC, USA, 2010. IEEE Computer Society.

[92] 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.

[93] The Mason Book: Escaping Substitutions. `http://www.masonbook.com/book/chapter-2.mhtml`.

[94] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[95] 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*, May 2010.

[96] Microsoft Live Labs. Microsoft Web Sandbox. `http://www.websandbox-code.org/Samples/genericSample.aspx`.

[97] Elinor Mills. Malicious Flash ads attack, spread via clipboard, August 2008. `http://news.cnet.com/8301-1009_3-10021715-83.html`.

[98] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. *Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

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

[100] XSS Prevention Cheat Sheet. `http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet`.

[101] PHP usage statistics. `http://www.php.net/usage.php`.

[102] Jeremy Pullicino. Google XSS Flaw in Website Optimizer Explained, December 2010. `http://www.acunetix.com/blog/web-security-zone/articles/google-xss-website-optimizer-scripts/`.

[103] William Robertson and Giovanni Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 283–298, Berkeley, CA, USA, 2009. USENIX Association.

[104] Ruby on Rails Security Guide. `http://guides.rubyonrails.org/security.html`.

[105] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[106] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.

[107] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Technical report, Microsoft Research, September 2010.

[108] Ben Schmidt. Google Analytics XSS Vulnerability. `http://spareclockcycles.org/2011/02/03/google-analytics-xss-vulnerability/`.

[109] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proceedings of the 15th international conference on financial cryptography and data security*, 2011.

[110] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[111] Jiwon Seo and Monica S. Lam. Invisitype: Object-oriented security policies, 2010.

[112] Smarty Template Engine: escape. `http://www.smarty.net/manual/en/language.modifier.escape.php`.

[113] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[114] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. 2006.

[115] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in fine. In *European Symposium on Programming*, pages 529–549, 2010.

[116] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *ICFP '11*. ACM, 2011.

[117] Nikhil Swamy, Brian Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2008.

[118] Nikhil Swamy, Joel Weinberger, Juan Chen, Ben Livshits, and Cole Schlesinger. Monadic refinement types for verifying javascript programs. Technical report, Microsoft Research, 2012. Microsoft Research Technical Report MSR-TR-2012-37.

[119] Template::Manual::Filters. `http://template-toolkit.org/docs/manual/Filters.html`.

[120] 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.

[121] TNW: The Next Web. YouTube hacked, Justin Bieber videos targeted. `http://thenextweb.com/socialmedia/2010/07/04/youtube-hacked-justin-bieber-videos-targeted/`.

[122] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP '10*. ACM, 2010.

[123] TwitPwn. DOM based XSS in Twitterfall. 2009.

[124] Twitter. All about the "onMouseOver" incident. `http://blog.twitter.com/2010/09/all-about-onmouseover-incident.html`.

[125] UTF-7 XSS Cheat Sheet. `http://openmya.hacker.jp/hasegawa/security/utf7cs.html`.

[126] Ashlee Vance. Times Web Ads Show Security Breach, September 2009. `http://www.nytimes.com/2009/09/15/technology/internet/15adco.html`.

[127] Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science*, 7(2:3), 2011.

[128] Wietse Venema. Taint support for PHP. ftp://ftp.porcupine.org/pub/php/php-5.2.3-taint-20071103.README.html, 2007.

[129] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42. Citeseer, 2007.

[130] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 1–16, New York, NY, USA, 2007. ACM.

[131] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, New York, NY, USA, 2007. ACM.

[132] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the International symposium on Software testing and analysis*, 2008.

[133] Joel Weinberger, Adam Barth, and Dawn Song. Towards client-side html security policies. In *Proc. of 6th USENIX Workshop on Hot Topics in Security*, 2011.

[134] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. An empirical analysis of xss sanitization in web application frameworks. Technical Report UCB/EECS-2011-11, EECS Department, University of California, Berkeley, Feb 2011.

[135] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks. In *Proc. of 16th European Symposium on Research in Computer Security (ESORICS)*, 2011.

[136] WhiteHat Security. WhiteHat Webinar: Fall 2010 Website Statistics Report. `http://www.whitehatsec.com/home/resource/presentation.html`.

[137] Wikipedia. Quirks mode — wikipedia, the free encyclopedia, 2011. [Online; accessed 18-March-2011].

[138] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, 2006.

[139] xssterminate. `http://code.google.com/p/xssterminate/`.

[140] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.

[141] Yii Framework: Security. `http://www.yiiframework.com/doc/guide/1.1/en/topics.security`.

[142] Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the web. In *WWW '09: Proceedings of the 18th international conference on World Wide Web*, pages 961–970, New York, NY, USA, 2009. ACM.

[143] M. Zalewski. Browser security handbook. *Google Code*, 2010.

[144] Zend Framework: Zend_Filter. `http://framework.zend.com/manual/en/zend.filter.set.html`.

[145] Kris Zyp. Secure Mashups with dojox.secure. `http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/`.