RETHINKING WEB PLATFORM EXTENSIBILITY

BY

MOHAN DHAWAN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Vinod Ganapathy

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2013

© 2013

MOHAN DHAWAN

ALL RIGHTS RESERVED

ABSTRACT OF THE DISSERTATION

RETHINKING WEB PLATFORM EXTENSIBILITY

by MOHAN DHAWAN

Dissertation Director: Vinod Ganapathy

The modern Web platform provides an extensible architecture that lets third party extensions, often untrusted, enhance and customize the Web browser and the Web applications. While the prevalence of extensions for both browsers and applications has been instrumental in making the Web browser hugely successful, there are two critical issues that the designers of the modern Web platform have not yet tackled in a principled manner. First, both the third party extensions and the extensible components of the Web platform include numerous vulnerabilities, which can compromise the security and privacy of end users. Second, the black-box and opaque nature of the Web platform limits the extent of extensibility achievable for Web developers, thereby hampering the development of novel browser-based user applications.

This dissertation develops new tools and techniques to address the problem of insecure extensibility in the Web platform, proposes novel *language* and *system* level solutions to make extensibility a first class primitive for developing Web software, and demonstrates that these methods are applicable to real-world Web applications and Web browser extensions.

Specifically, this dissertation makes the following three contributions. First, it studies and characterizes the problem of insecure JavaScript-based Web browser extensions using a specialized program analysis system, Sabre, which leverages JavaScript-level information flow mechanism to detect violations in client's confidentiality and integrity arising from execution of untrusted extensions. Second, it formalizes the concept of transactions for JavaScript and

implements Transcript, a language runtime system that allows hosting principals, i.e., Web browser and Web applications, to isolate untrusted JavaScript-based extensions using speculative execution. Lastly, this dissertation presents the design and implementation of Atlantis, a novel, extensible browser architecture that allows Web applications to define their own runtime environment and become more secure and robust. Atlantis enables developers with primitives to manage the Web application's security and privacy, and removes their dependence on opaque, legacy Web interfaces.

Acknowledgements

This dissertation would not have been possible without the contribution, encouragement, and guidance of a number of individuals.

I would like to thank my graduate advisor Professor Vinod Ganapathy and co-advisor Professor Liviu Iftode. Vinod has been a constant source of inspiration and guidance. His insights and feedbacks have directly shaped several ideas in this dissertation. His passion for excellence in research has motivated me to work harder and has greatly influenced my personality. I have learned tremendously from my interactions with Liviu, whose words of encouragement and wisdom have helped me throughout my graduate career. I would not have been successful in this endeavor without the help of my advisors. I would also like to thank Professor Ulrich Kremer and Dr. Kapil Singh (IBM Research) for their insightful comments to help improve this dissertation.

Over the past few years, I have also had the pleasure of working with several other outstanding people who have been instrumental in shaping my graduate career, and without their guidance this dissertation and several other research works would not have been successful. I would like to thank Professor Chung-chieh Shan (Indiana University) for enlightening me about the fundamentals of JavaScript, which has been central to this dissertation. I also had the honor and privilege of working closely with Professor Vern Paxson (UC Berkeley / ICSI Berkeley), Professor Renata Cruz Teixeira (LIP6 Paris), Dr. Christian Kreibich, Dr. Mark Allman and Dr. Nicholas Weaver (ICSI Berkeley). I have greatly benefited from their clear vision, infectious optimism, kind advice and insightful feedback. I am also thankful to Dr. James Mickens (Microsoft Research Redmond) and Dr. Úlfar Erlingsson (Google) for giving me an opportunity to work with them.

I have had the privilege of sharing my time at Rutgers with several excellent people. I want

to acknowledge the many members of the DiscoLab who over the past six years have contributed their valuable time, ideas, and opinions to several projects, meetings and practice talks that were crucial to me. I especially thank Aniruddha Bohra, Arati Baliga, Steve Smaldone, Pravin Shankar, Lu Han, Shakeel Butt, Rezwana Karim, Amruta Gokhale, Liu Yang and Nader Boushehrinejadmoradi.

I owe my deepest gratitude to my parents, my sister and my fiancée for their endless love and encouragement throughout this entire journey. My parents have always taken keen interest in my academic progress. It is because of their efforts that I had an excellent education and had the chance to make a career in computing. My sister, Roopam, taught me how to read and write, and without her dedication and affection I would not have reached this career milestone. I am also thankful to my fiancée, Sneha, for her patience while I was working on this dissertation.

I attribute everything I have ever achieved to my family, who have supported me even when I doubted myself. Without them I would have struggled to find the inspiration and motivation needed to complete this dissertation. I thank my family and dedicate this dissertation to them.

Dedication

To my family, for their endless support and encouragement.

To Nanaji and Sanjiv bhaiya, who are deeply missed.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation develops new tools and techniques to address the problems of extensibility in the Web platform, proposes novel *language* and *system* level solutions to make extensibility a first class primitive for developing Web software, and demonstrates that these methods are applicable to real-world Web applications and Web browser extensions.

1.1 Motivation

An *extensible* system is one that allows expanding the capabilities of the system without significant modifications to the underlying infrastructure. The modern Web browser has evolved into an extensible platform enabling clients and developers to enhance the functionality of both Web browser and Web applications. Clients install Web browser extensions from galleries hosted by browser vendors, like Google and Mozilla, to enhance the capabilities and appearance of their Web browsers. Both Google and Mozilla host over 10,000 Web browser extensions in their extension galleries, which experience several hundred thousand downloads daily [80,120]. In contrast, Web application developers leverage the extensibility in the HTML, CSS and JavaScript to integrate third party Web application extensions, such as advertisements, analytics, widgets and JavaScript-libraries, to develop rich, mashup applications.

Third party resources, often untrusted, such as browser and application extensions, provide necessary functionality to the users, but they can also be exploited to distribute malware. A study [52] of the Fortune 500 companies, Quantcast Top 1000 sites and other highly trafficked Web sites by Dasient, revealed that 75% of enterprises use third party JavaScript widgets on their Web sites, 42% display external advertisements, and up to 91% run outdated, third party applications, thereby greatly increasing their exposure to malware infections. Similarly, third

party browser extensions have been used as attack vectors [27, 32, 140] to install malware on the client's systems and steal sensitive information.

While the prevalence of extensions for both browsers and applications has been instrumental in making the browser hugely successful, there are two critical issues that the designers of the modern Web platform have not yet tackled in a principled manner. First, both the third party extensions and the extensible components of the Web platform include numerous vulnerabilities, which can compromise the security and privacy of end users. Second, the black-box and opaque nature of the Web platform limits the extent of extensibility achievable for Web developers, thereby hampering the development of novel browser-based user applications.

Retrofitting security solutions may help to secure vulnerable Web browser APIs and extension frameworks, however a careful redesign of the Web platform is required that considers extensibility as a primary attribute rather than an optional feature. In this dissertation, we develop novel solutions that leverage *operating system* principles to redesign the Web platform and *secure* and *enhance* its extensibility.

1.2 Securing Extensibility of the Web Platform

Third party extensions are often of unknown provenance and can even be downright malicious in nature. According to a study [153] by Symantec, the top extensions technologies, such as Java, Flash, ActiveX and JavaScript, together account for over 70% of all extension vulnerabilities. Drawing parallels between the operating systems and the Web browser worlds, we observe that Web platform extensions are akin to device drivers in an operating system. Just as a vulnerable device driver can be exploited to compromise the entire operating system, a buggy extension can be exploited by a remote attacker to take control over the entire Web browser. Thus, vulnerable and malicious extensions may lead to arbitrary code execution, privilege escalation, or even denial of service for the client.

JavaScript-based extensions are hugely popular due to the comparative ease of development, availability on vendor galleries, and simplicity in usage. Since, JavaScript-based extensions execute with the privileges of the hosting principal, i.e., Web browser and Web applications, untrusted Web platform extensions can trivially violate client's confidentiality and

integrity. JavaScript-based browser extensions run with the privileges of the Web browser, and thus, they can access all the resources that the browser can access, such as cookies, passwords, file system, etc. In contrast, JavaScript-based application extensions run with the privileges of the hosting Web applications, and have access to the session cookies, GUI events, and sensitive application data, like the DOM tree [6]. Several incidents involving the Web platform extensions, for browsers such as Firefox [24,25,136,146,147,166] and Google Chrome [27,32,140], and applications, like New York Times [127], demonstrate the risks posed by vulnerable and malicious extensions to a client's security and privacy.

We characterize the behavior of untrusted JavaScript-based browser extensions using a specialized program analysis tool, **Sabre** [56], which leverages information flow techniques. Sabre is a reference monitoring mechanism that implements JavaScript-level information flow analysis to identify offending flows, and protect against privacy violations due to browser extensions. The key idea is to identify and mark sensitive objects, track their propagation during program execution, and take appropriate action in the event of an offending information flow.

We also develop **Transcript** [57,58], a novel sandboxing mechanism, to address the issues of insecure JavaScript-based Web platform extensions. Transcript uses speculative execution to isolate all effects of unmodified, third party JavaScript code execution, and allows hosting principals to enforce fine-grained security policies to mediate all actions of the untrusted code. Transcript leverages first principles to implement *isolation* as a primitive for the JavaScript language, and thus, helps to secure extensibility of the Web platform. Unlike reference monitoring mechanisms that use access control policies or information flow control, which may allow undesirable effects to persist even after detecting and preventing a security policy violation, Transcript ensures that none of the effects of third party JavaScript code execution would be applied in the event of a security policy violation.

1.3 Enhancing Extensibility of the Web Platform

Web platform extensions, for both Web browsers and Web applications, invoke Web browser APIs to display and render content, and provide useful functionality for users. But, the Web

browser still represents a black-box, since the applications and extensions have very little control over the browser's execution environment. For example, since both Safari and Google Chrome use a common layout and rendering engine, Web applications running on these Web browsers exhibit similar rendering bugs [47], and even though the developers are aware of these issues, they cannot solve these problems themselves and must wait for the vendors to issue a fix. This restriction in extending or modifying the browser's execution environment is an artifact of the legacy, monolithic browser architecture, and severely limits the extent of achievable extensibility and functionality for the applications and extensions.

Moreover, the modern Web protocol is huge and complex, and has been standardized in volumes of specifications [50, 62, 74, 159, 165, 167–169, 171]. However, browsers implement these interfaces differently and present Web applications with different execution environments. Further, Web software developers must also account for different versions of the same browsers in use. For example, Internet Explorer 8 does not allow applications to install capture phase event handlers [118], while later versions follow the standards and implement the three phase event propagation model, which includes the capture phase.

Developers use application-level JavaScript frameworks, like jQuery [12], YUI [174], etc., which strive to provide browser-neutral interfaces for development. However, in reality, these frameworks cannot hide browser quirks or vagaries of different browser subsystems, thereby causing applications to behave and fail differently on different Web browsers. For example, recent versions of Firefox, Chrome, Safari, and Internet Explorer are vulnerable to a CSS parsing bug that allow an attacker to steal private data from authenticated Web sessions [90]. In another instance, Microsoft recently issued a patch [117] for Internet Explorer 8 that fixed a bug in the JSON parser. In both of these cases, Web developers who were aware of the security problems were reliant on browser vendors to implement fixes. Thus, writing secure and robust Web software in a browser-neutral way remains a challenge for the developers.

We leverage exokernel [67] principles to develop **Atlantis** [112], a novel, extensible browser architecture that provides unprecedented extensibility, and addresses the concerns about writing secure and robust browser-neutral Web software. The key idea is to allow Web

applications to define their own execution environment, with Atlantis supporting a narrow interface for executing the application-defined environment and responsible for implementing browser's security policies. Since applications in Atlantis *control* the execution environment, they can extend, introspect or modify the Web stack in whichever manner they want.

1.4 Contributions

Problem statement: Modern Web platforms offer *limited* extensibility without *sufficient* security.

Thesis statement: We can successfully apply operating system principles to *secure* and *enhance* the extensibility of the Web platform.

This dissertation supports the above problem and thesis statements and makes the following contributions:

- We study and characterize the nature of JavaScript-based Web browser extensions (Chapter 4). We present Sabre, an in-browser JavaScript-level dynamic information flow mechanism to detect violations in client's confidentiality and integrity arising from execution of untrusted JavaScript-based browser extensions.

- We present novel *language* and *system* level approaches to secure and enhance Web platform extensibility by implementing extensibility as a first class primitive for the Web platform. Transcript (Chapter 5) implements a language runtime system to provide fine-grained security for Web application extensions using transactions for JavaScript. Atlantis (Chapter 6) is a novel extensible Web browser architecture that enables Web applications to leverage the browser's extensibility to become more secure and robust. While these systems were targeted towards Web application extensions, they are equally applicable for Web browser extensions.

- We formalize the notion of transactions for JavaScript and implement Transcript, which allows Web applications to speculatively execute untrusted JavaScript code by enclosing them in transactions (Chapter 5). We enhance the JavaScript language with a

`transaction` primitive, provide runtime support for speculative JavaScript and DOM updates, implement suspend/resume feature for JavaScript to ease introspection of transaction state and policy enforcement, and design novel strategies to implement transactions in commodity JavaScript interpreters.

- We design and implement Atlantis, a novel, exokernel-based Web browser architecture that allows each Web application to define its own runtime environment (Chapter 6). Atlantis enables Web developers with primitives to completely control the application security and privacy, and removes their dependence on opaque legacy Web interfaces. We also discuss the various engineering choices and several key optimization techniques enabling Atlantis to be applicable for real-world Web applications.

- We report on extensive experiments with these tools, including examples of previously undiscovered information flow violations in popular browser extensions using Sabre (Chapter 4), the effectiveness and cost of speculatively executing JavaScript using Transcript (Chapter 5) and flexibility of user-defined Web stack using Atlantis (Chapter 6).

1.5 Statement of Joint Work

The following is a list of people who co-authored papers from which material was used in this dissertation. Chapter 5 of this dissertation is the result of a collaboration between by my advisor, Professor Vinod Ganapathy, and Professor Chung-chieh Shan. During this time, Professor Shan contributed to the design of the speculative execution mechanism in Transcript. The exokernel browser presented in Chapter 6 was jointly designed and developed with Dr. James Mickens (Microsoft Research Redmond). He implemented the Syphon interpreter that was part of the Atlantis Web browser kernel.

Chapter 2

The Web Ecosystem

Modern Web browsers have evolved into sophisticated computational platforms and this success has spawned a mighty ecosystem of Web technologies and standards. In this chapter, we provide a brief background about these Web technologies and their interaction with the different browser subsystems.

2.1 Core Web Application Technologies and JavaScript

Modern Web development uses four essential technologies: HTML, CSS, JavaScript, and content plugins. HTML [171] is a declarative markup language that describes the basic content in a Web page. HTML defines a variety of tags for including different kinds of data. For example, an `<img>` tag references an external image, and a `<b>` tag indicates a section of bold text. Tags nest using an acyclic parent-child structure. Thus, a page's tags form a tree, also known as the DOM tree [167], which is rooted by a top-level `<html>` node.

Using tags like `<b>`, HTML supports rudimentary manipulation of a page's visual appearance. However, cascading style sheets (CSS) [168] provide much richer control. Using CSS, a page can choose fonts and color schemes, and specify how tags should be visually positioned with respect to each other.

JavaScript [76] is the *de facto* language for client-side browser scripting. It allows Web pages to dynamically modify their HTML structure and register handlers for GUI events. JavaScript also allows a page to asynchronously fetch new data from Web servers.

JavaScript has traditionally lacked access to client-side hardware like Web cameras and microphones. However, a variety of native code plugins like Flash, Java and Silverlight provide access to such resources. These plugins run within the browser's address space and are often

used to manipulate audio or video data. A Web page instantiates a plugin using a special HTML tag like `<object>`.

2.2 Web Browser

Browsers implement the core Web technologies using a standard set of software components. The experience of "Web browsing" on a particular browser is largely governed by how the browser implements these standard modules.

- The *network stack* implements various transfer protocols, like `http://`, `https://`, `ftp://` and `file://`.
- The *HTML* and *CSS* parsers validate a page's HTML and CSS content. Since malformed HTML and CSS are pervasive, parsers define rules for coercing ill-specified pages into a valid format.
- The browser internally represents the Web page using a data structure called the *DOM tree*. "DOM" is an abbreviation for the Document Object Model, a browser-neutral standard for describing HTML content [167]. The DOM tree contains a node for every HTML tag. Each node is adorned with the associated CSS data, as well as any application-defined event handlers for GUI activity.
- The *layout and rendering engine* traverses the DOM tree and determines the visual size and spatial positioning of each element. For most complex Web pages, the layout engine requires multiple passes over the DOM tree to calculate the associated layout.
- The *JavaScript interpreter* implements the core JavaScript runtime, which defines basic datatypes like strings, and provides simple library services like random number generation. The interpreter also reflects the DOM tree into the JavaScript namespace, defining JavaScript objects which are essentially proxies for internal browser objects defined in *native code* (typically C++).
- The *storage layer* manages access to persistent data like cookies, cached Web objects, and DOM storage [159], an HTML5 abstraction that provides each domain with several

megabytes of key/value storage.

Even simple browser activities require numerous communications between the modules described above. For example, let us assume that a JavaScript code wants to dynamically add an image to a page. Then, the JavaScript interpreter must request the browser core to send a fetch request to the network stack. Once the stack has fetched the image, it examines the response headers and caches the image, if necessary. The browser adds a new image node to the DOM tree, recalculates the layout, and renders the result. The updated DOM tree is then reflected into the JavaScript namespace, and the interpreter triggers any application-defined event handlers that are associated with the image load.

Part I

Securing Web Platform Extensibility

Chapter 3

JavaScript-based Extensibility in the Web Platform

JavaScript-based extensions for the Web platform are hugely popular. In this chapter, we give a high-level overview of the security issues concerning such Web platform extensions.

3.1 Web Application Extensions

Extensibility in the HTML and JavaScript languages allows Web developers to integrate third party JavaScript-based Web application extensions, such as advertisements, widgets and JavaScript libraries, to develop rich, mashup Web applications. Such Web application extensions can *easily* be integrated in the hosting Web application using HTML tags, such as `<script>` and `<iframe>`, or dynamically loaded using JavaScript constructs, such as `document.write` and `innerHTML`. But, this ease of integration is a major source of concern because the *same-origin policy* [158] enforces that the third party JavaScript gains complete access to the hosting Web application's resources, including the application's DOM and other sensitive JavaScript objects. Over the years, numerous incidents involving malicious, third party JavaScript code that abuse the *same-origin policy* have been reported. For example, the self-propagating worm "Samy" leveraged a combination of cross-site scripting and lax security in the Internet Explorer Web browser to affect over a million users [3].

While several mechanisms to sandbox untrusted JavaScript content have been proposed in the past, none of these mechanisms provide a flexible, yet comprehensive, solution to securely include untrusted JavaScript code. HTML `<iframe>`s provide a rigid, process-like isolation abstraction for Web applications, while other approaches to statically verify the behavior of third party code [49, 72, 75, 106, 108, 109] often require the use of subsets of JavaScript. Thus they are not compatible with legacy JavaScript code, which may include dynamic constructs,

such as `this`, `with` and `eval`. While `<iframe>`s are useful, they are not practical in situations where the Web application and the third party JavaScript-based application extensions must interact or access each others JavaScript objects. For example, JavaScript frameworks, contextual advertisements and even widgets, like spell checkers or code syntax highlighters, may need to access the Web application's DOM.

3.2 Web Browser Extensions

Web browser extensions are mostly written using open technologies like HTML, CSS and JavaScript, just like normal Web applications. However, unlike JavaScript executing within a Web page (also known as *content script*), browsers execute extension JavaScript (also known as *chrome script*) with elevated privileges to enable them to access privileged browser APIs and perform useful tasks.

Mozilla was the first vendor to introduce JavaScript-based extension framework for its suite of applications, including the Firefox Web browser. But, this legacy extension architecture has a number of features that make it vulnerable. We briefly discuss some of them below:

- ***Unified JavaScript heap***: Mozilla's legacy extension development environment provides a unified JavaScript heap for all JavaScript code execution. Therefore, both privileged *chrome scripts* and unprivileged *content scripts* reside in the same heap, raising the risk of shared references. Mozilla uses XrayWrappers (also know as XPCNativeWrappers [125]) to isolate the untrusted references of the *content* JavaScript from the *chrome* JavaScript, but this mechanism has limitations and a history of exploitable bugs [35,121]. Such scenarios have previously been used to exploit vulnerable extensions [136, 166].

- ***Privileged objects***: All *chrome scripts* have default access to the global `window` object and its properties. The `Components` object is a special property of the `window` that provides access to the browser's sensitive XPCOM APIs. Thus, if an attacker gets a reference to the `Components` object, he effectively has control over the entire browser. The fact that the `Components` object is so powerful and is yet available to all scripts by default significantly increases chances of vulnerability exploitation in a shared heap environment.

- ***Global access***: One consequence of having a unified heap for JavaScript execution is that top-level objects declared in *chrome scripts* are attached as properties of the global object. This often results in namespace collisions across different extensions or even different *chrome scripts* within the same extension. Further, since globals defined in one script can be accessed and modified from another script, their careless use in asynchronous executions might also lead to data races.

- ***Chrome DOM***: Much as the DOM API available to *content scripts* on a Web page, *chrome scripts* also have access to the chrome DOM. The chrome DOM is responsible for the visual representation of the browser's UI including toolbars, menus, statusbar, icons and context menus. Since much of Firefox's UI is also written in JavaScript, *chrome scripts* can programmatically access and modify the browser's entire UI.

Apart from the issues discussed above, there is another significant factor that makes Mozilla's legacy extension architecture vulnerable. Since part of the Mozilla platform is written in JavaScript with some other core components written in C/C++, legacy extensions written in the same language (i.e., JavaScript) as the platform they run on are particularly dangerous because there is no language boundary (*i.e.,* a language-based isolation mechanism) to separate various components of the browser. Without a language boundary, a concerted effort is required to restrict access to critical functionality, but Mozilla's legacy extension architecture only makes a weak attempt to do so.

There is much prior work [34, 56, 59] highlighting the shortcomings of Mozilla's legacy extension architecture. Thus, both Mozilla and Google have developed new extension architectures that enable developers to build extensions that are secure by design. We briefly highlight the salient features of these new extension platforms.

3.2.1 Google Chrome Extension Platform

Google Chrome implemented a secure extension framework [35] based on the principle of least privilege, privilege separation, and strong isolation to protect users from vulnerabilities in benign but vulnerable extensions. Figure 3.1 shows the overall architecture of a Google Chrome extension. Each extension is divided into three process-isolated components – namely

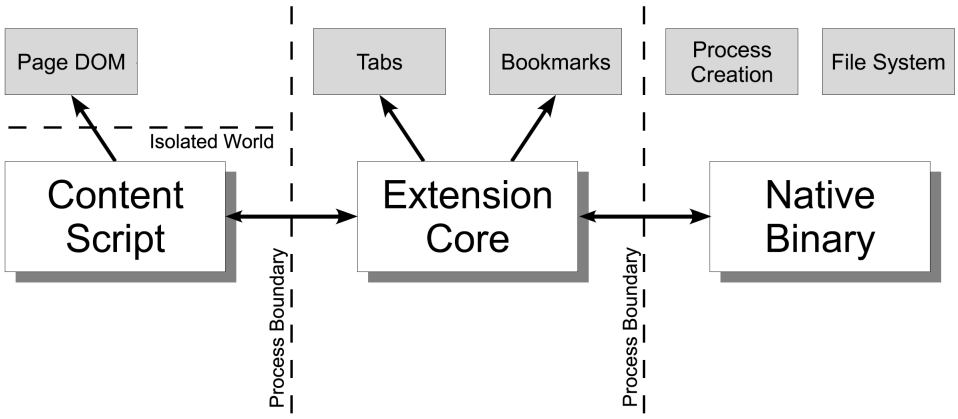


Figure 3.1: **Architecture of a Google Chrome extension.** Adapted from [35].

content scripts, *extension core* and a *native binary* – with each component progressively having more access to the privileged browser APIs and less exposure to potentially malicious Web content. A *content script* can interact with Web content but cannot access any browser APIs. The *extension core* has access to all the browser APIs but cannot directly interact with Web content, while the optional *native binary* implements the NPAPI interface [17] to interact with the *extension core* and has access to the entire system.

The Google Chrome extension platform implements three main security features:

- ***Privilege separation***: A Google Chrome extension is a collection of *content scripts* and an *extension core*. While the *content scripts* can access the Web content, only the *extension core* has access to the privileged Google Chrome extension API. As shown in the Figure 3.1, *content scripts* and *extension core* are isolated using separate processes. This separation of privileges prevents a malicious Web attacker to gain direct access to privileged browser APIs.

- ***Isolated worlds***: Isolated worlds is a mechanism to execute a *content script* with its *own* copy of JavaScript and DOM objects, instead of sharing references to these objects with a potentially malicious Web content. Since the *content scripts* and Web pages never exchange JavaScript objects, it becomes harder for malicious Web content to tamper with the content scripts.

- ***Permissions***: Unlike legacy Firefox extensions, which allow access to *all* browser APIs,

Google Chrome extensions, by default, do not have access to any browser API. Each extension must explicitly request for the permissions to access the desired browser APIs. Further, as mentioned earlier, only the *extension core* is privileged enough to request access to the browser APIs. *Content scripts* can only interact with the Web content and send messages to the *extension core*.

3.2.2 Mozilla Jetpack

The Jetpack extension framework [11, 16] is a recent effort by Mozilla to incorporate security principles in the design of the extension architecture, thereby improving the overall security of extensions. Jetpack uses a layered defense architecture to make it harder for an attacker to compromise extensions, and limit the damage done if he succeeds in compromising all or part of the extension. The Jetpack project shares ideological similarities with the Google Chrome extension architecture [35]. It has also been motivated in part by the new multi-process Firefox architecture [124].

Conceptually, each Jetpack extension has three parts: (i) at least one add-on script (also known as *chrome script*), (ii) zero or more *content scripts*, and (iii) a set of *core modules*, which have access to the sensitive browser APIs. The *chrome script* executes within the Web browser with restricted but elevated privileges, while the *content scripts* interact with the Web page and are unprivileged. By default, the *chrome script* does not have direct access to any the sensitive browser APIs, except for those that it explicitly requested. The Jetpack framework offers the following new features:

- ***Chrome/content heap partitioning***: The multi-process Firefox architecture mandates that both the *chrome* and *content scripts* execute in separate processes. This partitioning guarantees isolation of the JavaScript heap for the *chrome* and *content scripts* and prevents inadvertent access by *content scripts* to privileged object references in the *chrome* code. Communication amongst the *chrome* and *content scripts* is made possible through IPC with all messages exchanged in the standard JSON format.

- ***Content script integrity***: Although *chrome/content* heap partitioning ensures that *content scripts* can never reference privileged resources, it alone does not ensure the integrity of

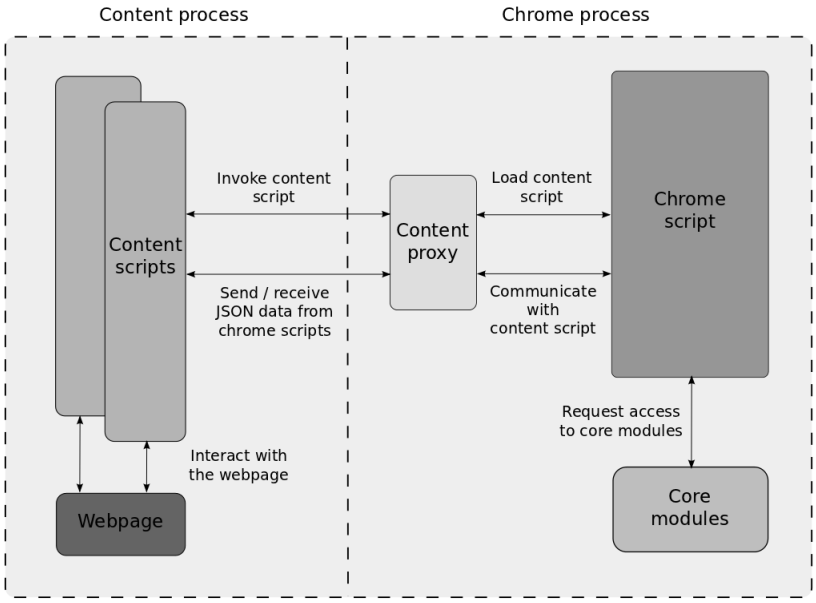


Figure 3.2: **Architecture of a simple Jetpack extension.**

Jetpack's *content scripts*. This is because *content scripts* execute in the context of the Web page, and a malicious Web page can redefine objects referenced by the *content script*. Jetpack uses *content proxies* to protect the integrity of *content scripts*. Content proxies allow the *content script* to access the content on the Web page while still having access to the native objects and APIs, even if the Web page has redefined them.

- ***Chrome privilege separation***: Mozilla's legacy extension architecture has two major drawbacks from a security viewpoint. First, all *chrome scripts* have elevated privileges and unrestricted access to all browser APIs. This means that any breach in the extension security would yield browser-level privileges to the attacker. Second, the extension developer is responsible for handling all the sensitive browser APIs.

  The Jetpack framework addresses both drawbacks. Jetpack provides developers with a set of *core modules* that encapsulate the functionality of the privileged browser APIs, thus preventing inadvertent misuse of these APIs by the developer. Further, developers must explicitly request these *core modules* as required by the extension's *chrome scripts*. This restricts the set of privileges that an attacker can obtain in the event of a breach to only those requested by the exploited script.

The Jetpack framework further recommends developers to organize extensions as a hierarchy of user modules, each of which may itself request other user modules and zero or more *core modules* using the `require` interface. The set of privileges thus acquired by each user module is determined statically by analyzing the source code and enforced by the framework at runtime. The Jetpack framework further provides isolation among user and *core modules*. Objects declared within a module are local to the module unless exported explicitly through the module's `exports` interface.

Figure 3.2 shows the overall architecture of a Jetpack extension. Together with process separation and use of content proxies, the Jetpack framework ensures that *chrome* and *content* scripts execute as intended by the developer. Privilege separation within *chrome scripts* further limits ensuing damage in case of vulnerabilities exploited within the extension.

Chapter 4

Characterizing JavaScript-based Web Browser Extensions

In this chapter, we present a specialized program analysis system, Sabre (Security Architecture for Browser Extensions), to detect and prevent violations in client's confidentiality and integrity due to untrusted Web browser extensions. Sabre is a dynamic information flow analyzer for JavaScript, which we use to study and characterize the behavior of third party JavaScript-based browser extensions (JSEs) for the Firefox Web browser. We present Sabre in detail—examining its design and implementation in a commodity Web browser, the heuristics used to achieve precision, and evaluate its effectiveness on several popular and malicious real-world browser extensions.

4.1 Problem

Modern Web browsers support an architecture that lets third party JSEs enhance the core functionality of the browser. To allow easy access to browser resources and to support a rich set of functionalities, browsers execute JSEs with elevated privileges. However, doing so renders the browser susceptible to attacks via JSEs. Malicious JSEs may exploit elevated privileges to steal sensitive data or snoop on user activity. Worse, benign JSEs from trusted vendors may contain vulnerabilities that, in combination with browser vulnerabilities, may be exploited by remote attackers. The problem is exacerbated by the lack of good environments and tools, such as static bug finders, for code development in JavaScript. Moreover, because subtle bugs only manifest when a JSE is used with certain versions of the browser, comprehensive testing of JSEs for security vulnerabilities is often difficult.

Recent attacks [32, 140] confirm that JSEs pose a threat to browser security, and there are two critical factors that contribute to this threat:

(1) Inadequate sandboxing of JavaScript in a JSE. Unlike JavaScript code in a Web application, which executes with restricted privileges [19], JavaScript code in a JSE executes with the privileges of the browser. JSEs are not constrained by the same-origin policy [145], and can freely access sensitive entities, such as the cookie store, browsing history and file-system. Importantly, *these features are necessary* to create expressive JSEs that support a rich set of functionalities. For example, JSEs that provide cookie/password management functionality rely critically on the ability to access the cookie/password stores.

However, JSEs from untrusted third parties may contain malicious functionality that exploits the privileges that the browser affords to JavaScript code in an extension. Examples of such JSEs exist in the wild. They are extremely easy to create and can avoid detection using stealth techniques [23, 25–27, 37, 104].

(2) Browser and JSE vulnerabilities. Even if a JSE is not malicious, vulnerabilities in the browser and in JSEs may allow a malicious Web site to access and misuse the privileges of a JSE [24, 136, 146, 147, 166]. Vulnerabilities in older versions of Firefox/Greasemonkey allowed a remote attacker to access the file system on the host machine [136, 166]. Similarly, vulnerabilities in Firebug [24, 146] allowed remote attackers to execute arbitrary commands on the host machine using exploits akin to cross-site scripting.

As mentioned earlier in Section 3.2, both Google and Mozilla have introduced new extension frameworks [16, 35] that provide safeguards against the first class of risks by isolating the execution of JavaScript code on the Web page (unprivileged) from the JavaScript code executing within the extension (privileged). Although this isolation of privileged and unprivileged code limits the threats posed by a Web attacker by disallowing direct access to sensitive browser APIs, majority of the threats posed by vulnerable browser APIs and JSEs still persists. Moreover, these modern frameworks are useful only for extensions developed from scratch. There are about 10,000 legacy extensions with over 500 million instances of these extensions in use daily [120], so both the above mentioned threats continue to linger.

4.2 Motivating Examples

We now briefly describe four real-world examples, both benign but vulnerable, and malicious JSEs. These examples clearly demonstrate that the threat of untrusted JSE execution, in both legacy and modern frameworks, is very real, and untrusted JSE execution can have severe repercussions on a client's security and privacy.

(1) Greasemonkey/Firefox Vulnerability

Greasemonkey is a popular JSE that allows user-defined scripts to make changes to Web pages on the fly. For example, a user could register a script with Greasemonkey that would customize the background of Web pages that he visits. Greasemonkey exports a set of APIs (prefixed with "`GM`") that user-defined scripts can be programmed against. These APIs execute with elevated privileges because user-defined scripts must have the ability to read and modify arbitrary Web pages. For example, the `GM_xmlhttpRequest` API allows a user-defined script to execute an `XMLHttpRequest` to an arbitrary Web domain, and is not constrained by the same-origin policy.

Unfortunately, a combination of vulnerabilities in older versions of Greasemonkey (CVE-2005-2455) and Firefox (CVE-2006-1734) allowed scripts on a Web page to capture references to `GM` API functions (`GM_xmlhttpRequest` in particular) using the JavaScript `watch` function, as shown in Figure 4.1. When the page loads, the script uses this reference to issue a `GET` request to read the contents of the `boot.ini` file from the local file system. Although the script in Figure 4.1 simply modifies the DOM to store the contents of the `boot.ini` file, it could instead use a `POST` to transmit this data over the network to a remote attacker.

This example illustrates how a malicious Web site can exploit JSE/browser vulnerabilities to steal confidential user data.

```
1  <script type="text/javascript">
2    window._GM_xmlhttpRequest = null;
3    function trapGM(...)  {
4      window._GM_xmlhttpRequest = window.GM_xmlhttpRequest;
5      ...
6    }
7    function checkGM() {
8      if (window._GM_xmlhttpRequest) {
9        window._GM_xmlhttpRequest({
10         method:  'GET', url:  'file:///c:/boot.ini',
11         onload:  function(Response) {
12           document.formname.textfield.value =
                              Response.responseText;
13       }});
14     }
15   }
16   if (typeof window.addEventListener != 'undefined') {
17     window.watch('GM_apis', trapGM);
18     window.addEventListener('load', checkGM, true);
19   }
20 </script>
```

Figure 4.1: **Example of malicious JavaScript code that exploits the Greasemonkey vulnerability to read sensitive contents from the file system.** This snippet reads the contents of `boot.ini` from disk (adapted from [136]).

(2) Firebug Vulnerabilities

Firebug is a popular JSE that provides a development and debugging environment for HTML, CSS and JavaScript code. As a code development aid, Firebug exports a `console` interface that scripts loaded in the browser can use to display messages within the Firebug console. For example, a script on a Web page could include `console.log({'<html>Hello world</html>'})`, which would in turn display this message in the console. Because the Firebug console executes with `chrome` privileges, Firebug sanitizes the inputs received from the `console` interface, *e.g.,* it escapes special characters in the arguments to `console.log`.

However, input sanitization vulnerabilities in an older version of Firebug (CVE-2007-1878, CVE-2007-1947) [24, 146] allowed a malicious Web page to inject JavaScript code into the Firebug console. Although this attack is similar in flavor to XSS attacks, it can cause more damage because the injected code executes with `chrome` privileges. `chrome` here refers to a privilege level within the Firefox browser, and the code running with chrome privileges is

```
1  const {Cc, Ci} = require("chrome");
2  let Preferences = {
3    _branches:  {},
4    _caches:  {},
5    getBranch:  function (name) {
6      if (name in this._branches) return this._branches[name];
7      let branch = Cc["@mozilla.org/preferences-service;1"]
         .getService(Ci.nsIPrefService).getBranch(name);
8      .../* other statements */
9      return this._branches[name] = branch;
10   }, ...  /* other properties */
11 };
12 exports.Preferences = Preferences;
```

Figure 4.2: **Code snippet of a module from a real-world Jetpack that leaks the capability to access and modify browser preferences.**

allowed to do everything, unlike the Web content, which is restricted in several ways. For example, the injected JavaScript code could invoke the `nsIProcess` or `nsILocalFile` interfaces exported by XPCOM and start a process or read/modify the contents of a file on the local host, thereby affecting both the confidentiality and integrity of the host. In contrast, code injected into a Web application in an XSS attack is bound by the same-origin policy and can only access data (*e.g.,* cookies) belonging to the vulnerable Web application's domain.

(3) Customizable Shortcut Capability Leak

Although Mozilla's Jetpack framework implements the principle of least authority for each extension module, it does not safeguard against developer mistakes, such as those where developers request more privileges than required, with the result that unintended capability leaks are frequent. Such capability leaks can be used by remote attackers to violate a client's confidentiality and integrity.

Consider the code snippet as shown in Figure 4.2, which represents the actual code of the `Preferences` module from 'Customizable Shortcuts' [4], a popular Jetpack with over 14,500 users. This module exports a method `getBranch` that inadvertently enables access to the browser's entire preference tree. If another module imported the `Preferences` module, it would receive additional capabilities to access and modify the user's preferences for all extensions *without explicitly requiring access to the user preferences*; in effect, the importing

```
1  function do_sniff() {
2    var hesla =
         window.content.document.getElementsByTagName("input");
3    data = "";
4    for (var i = 0; i < hesla.length; i++) {
5      if (hesla[i].value != "") {
6        ...
7        data += hesla[i].type + ":" + hesla[i].name
                     + ":" + hesla[i].value + "";
8        ...
9      }
10   }
11   // the rest of the code sends 'data' via an email message.
12 }
```

Figure 4.3: **A snippet of code from FFsniFF, a malicious JSE.**

module would become over-privileged.

(4) A Malicious JSE

FFsniFF (Firefox Sniffer) [25] is a malicious JSE that, if installed, attempts to steal user data entered in HTML forms. When a user "submits" an HTML form, FFsniFF iterates through all non-empty `input` fields in the form, including password entries, and saves their values. It then constructs SMTP commands and transmits the saved form entries to the attacker (the attack requires the vulnerable host to run an SMTP server). FFsniFF also attempts to hide itself from the user by exploiting a vulnerability in the Firefox extension manager (CVE-2006-6585) to delete its entry from the add-ons list presented by Firefox.

Figure 4.3 presents a simplified snippet of code from FFsniFF and illustrates the ease with which malicious extensions can be written.

4.3 Our Approach: JavaScript-level Information Flow Tracking

All the examples discussed in Section 4.2 involve unprivileged attackers accessing privileged browser interfaces. Such scenarios provide a compelling case to leverage information flow tracking for detecting and preventing confidentiality and integrity violations in clients. To do so, we have implemented Sabre, which leverages *in-browser information flow tracking* to analyze

JSEs.

Sabre associates each in-memory JavaScript object with a label that determines whether the object contains sensitive information. Sabre modifies this label when the corresponding object is modified by JavaScript code (contained both in JSEs and Web applications). Sabre raises an alert if a JavaScript object containing sensitive data is accessed in an unsafe way, *e.g.,* if a JSE attempts to send a JavaScript object containing sensitive data over the network or write it to a file. In addition to detecting such confidentiality violations, Sabre also uses the same mechanism to detect integrity violations, *e.g.,* if a JSE attempts to execute a script received from an untrusted domain with elevated privileges. The rest of the chapter discusses Sabre's design and implementation in detail.

4.3.1 Sabre in Action

We now revisit the four attack scenarios described in Section 4.2 and describe how Sabre can effectively thwart such attacks.

(i) Detect confidentiality violations. Information flow tracking as implemented in Sabre detects confidentiality violations, due to the Greasemonkey/Firefox vulnerability, when sensitive user data (`boot.ini`) is accessed in unsafe ways. In particular, Sabre marks as sensitive all data that a JSE reads from a pre-defined set of sensitive sources, including the local file system. The call to `window._GM_xmlhttpRequest` (line 9 in Figure 4.1) executes JavaScript code from Greasemonkey to access the local file system. Consequently, `Response.responseText`, which this function returns, is also marked sensitive. In turn, the DOM node that stores this data is also marked as sensitive because of the assignment on line 12. Sabre raises an alert when the browser attempts to send contents of the DOM over the network, *e.g.,* when the user clicks a "submit" button.

The above example also illustrates the need to precisely track security labels across browser subsystems. For instance, Sabre detects the above attack because it also modifies the browser's DOM subsystem to store labels with DOM nodes. Doing so allows Sabre to determine whether a sensitive DOM node is transmitted over the network. An approach that only tracks security labels associated with JavaScript objects (*e.g.,* [29, 157]) will be unable to precisely detect this

attack.

(ii) Detect integrity violations. Much as prior work has used JavaScript-level taint tracking to detect XSS attacks [157], Sabre can also detect script injection attacks in Firebug. In particular, Sabre considers all data received from the `console` interface as untrusted because this interface is exposed to Web applications. Sabre would report an alert when the `nsILocalFile` or `nsIProcess` interface is invoked with untrusted parameters that are derived from data received through the `console` interface. Sabre differs from prior work [157] because it must also reason about information received from a number of cross-domain interfaces, such as access to the file system and the network, that are not accessible to JavaScript code in Web applications.

(iii) Detect capability leaks. Dynamic JavaScript-level information flow tracking as implemented in Sabre can easily detect capability leaks in JSEs. Since Sabre marks data that a JSE reads from XPCOM sources as sensitive, the assignment in line 7 in Figure 4.2 causes `branch` to become privileged. When the method in line 9 returns this privileged instance to a caller, Sabre can detect capability leak from the Jetpack module and raise an alert.

(iv) Defend against malicious JSEs. Sabre protects against FFsniFF because it considers all data received from form fields on a Web page as sensitive. This sensitive data is propagated to both the array `hesla` and the variable `data` via a series of assignment statements. Sabre raises an alert when FFsniFF attempts to send the contents of the sensitive `data` variable along with SMTP commands over an output channel (a low-sensitivity sink) to the SMTP server running on the host machine.

4.3.2 Inadequacies of Prior Techniques

While there is much prior work on the security of untrusted browser extensions such as plugins and BHOs (which are distributed as binary executables) particularly in the context of spyware [64, 97, 101], there is relatively little work on analyzing the security of JSEs. Existing techniques to protect against an untrusted JSE rely on load-time verification of the integrity of the JSE, *e.g.,* by ensuring that scripts are digitally signed by a trustworthy source. However,

such verification is agnostic to the code in a JSE and cannot prevent attacks enabled by vulnerabilities in the browser or the JSE. Ter-Louw *et al.* [104] developed a runtime agent to detect malicious JSEs by monitoring XPCOM calls and ensuring that these calls conform to a user-defined security policy. Such a security policy may, for instance, prevent a JSE from accessing the network after it has accessed browsing history. Unfortunately, XPCOM-level monitoring of JSEs is too coarse-grained and can be overly restrictive. For example, one of their policies disallows XPCOM calls when SSL is in use, which may prevent some JSEs from functioning in a `https` browsing session. XPCOM-level monitoring can also miss attacks, *e.g.,* a JSE may disguise its malicious actions so that they appear benign to the monitor (in a manner akin to mimicry attacks [160]).

4.4 Tracking Information Flow with Sabre

This section describes the design and implementation of Sabre. We had three goals:

1. **Monitor all JavaScript execution.** Sabre must monitor all JavaScript code executed by the browser. This includes code in Web applications, JSEs, as well as JavaScript code executed by the browser core, *e.g.,* code in browser menus and XUL elements [22].

   Monitoring all JavaScript code is important for two reasons. First, an attack may involve JavaScript code in multiple browser subsystems. For example, a malicious JSE may copy data into a XUL element, which may then be read and transmitted by JavaScript in a Web application. In such cases, it is important to track the flow of sensitive data through the JSE to the XUL element and into the Web application. Second, JSEs may often contain code, such as scripts in XUL overlays, that may be included into the browser core. Such code often interacts with JavaScript code in a Web application. For example, an overlay may implement a handler that is invoked in response to an event raised by a Web application. It is key to track information flows through code in overlays because overlays from untrusted JSEs may be malicious/vulnerable.

2. **Ease action attribution.** When Sabre reports an information flow violation by a JSE, an analyst may need to determine whether the violation is because of an attack or whether

the offending flow is part of the advertised behavior of the JSE. In the latter case, the analyst must whitelist the flow. For example, PwdHash [144] is a JSE that scans and modifies passwords entered on Web pages. This behavior may be considered malicious if performed by an untrusted JSE. However, an analyst may choose to trust PwdHash and whitelist this flow. To do so, it is important to allow for easy action attribution, *i.e.,* an analyst must be able to quickly locate the JavaScript code that caused the violation and determine whether the offending flow must be whitelisted.

3. **Track information flow across browser subsystems.** JavaScript code in a browser and its JSEs interacts heavily with other subsystems, such as the DOM and persistent storage, including cookies, saved passwords, and even the local file system. Sabre must precisely monitor information flows across these subsystems because attacks enabled by JSEs (*e.g.,* those illustrated in Section 4.2) often involve multiple browser subsystems.

4.4.1 Security Labels

Sabre associates each in-memory JavaScript object with a pair of security labels. One label tracks the flow of sensitive information while the second tracks the flow of low-integrity information (to detect, respectively, violations of confidentiality and integrity). We restrict our discussion to tracking flows of sensitive information because confidentiality and integrity are largely symmetric.

Each security label stores three pieces of information: (i) a sensitivity level, which determines whether the object associated with the label stores sensitive information; (ii) a Boolean flag, which determines whether the object was modified by JavaScript code in a JSE; and (iii) the name(s) of the JSE(s) and Web domains that have modified the object. The sensitivity level is used to determine possible information flow violations, *e.g.,* if data derived from a sensitive source is written to a low-sensitivity sink. However, Sabre raises an alert only if the object was modified by a JSE. In this case, Sabre reports the name(s) of the JSE(s) that have modified the object. For example, in Figure 4.1, the DOM node that stores the response from the `_GM_xmlhttpRequest` call is marked sensitive. Further, the data contained in the

node is modified by executing code contained in Greasemonkey, via the return value from `_GM_xmlhttpRequest`. Consequently, Sabre raises an alert when the browser attempts to transmit the DOM node via HTTP, *e.g.,* when the user submits a form containing this node.

Sabre's policy of raising an alert only when an object is modified by a JSE is key to avoiding false positives. Recall that Sabre tracks the execution of *all* JavaScript code, including code in Web applications and in the browser core. Although such tracking is necessary to detect attacks via compromised/malicious files in the browser core, *e.g.,* overlays from malicious JSEs, it can also report confidentiality violations when sensitive data is accessed in legal ways, such as when JavaScript in a Web application accesses cookies. Such accesses are sandboxed using other mechanisms, *e.g.,* the same-origin policy. We therefore restrict Sabre to report an information flow violation only when a sensitive object modified by JavaScript code in a JSE (or overlay code derived from JSEs) is written to a low-sensitivity sink.

Security labels in Sabre allow for fine-grained information flow tracking. Sabre associates a security label with each JavaScript object, including objects of base type (*e.g.,* `int`, `bool`), as well as with complex objects such as arrays and compound objects with properties. For complex JavaScript objects, Sabre associates additional labels, *e.g.,* each element of an array and each property of a compound object is associated with its own security label. In particular, an object `obj` and its property `obj.prop` each have their own security label.

Sabre stores security labels by directly modifying the interpreter's data structures that represent JavaScript objects. Doing so considerably eases the design of label propagation rules for a prototype-based language such as JavaScript. A JavaScript object inherits all the properties of its ancestor prototypes. Therefore, an object's properties may not directly be associated with the object itself. For example, an object `obj` may access a property `obj.prop`, which in turn may result in a chain of lookups to locate the property `prop` in an ancestor prototype of `obj`. In this case, the sensitivity-level of `obj.prop` is the sensitivity of the value stored in `prop`. Sabre stores the label of the property `prop` with the in-memory representation of `prop`. Its label can therefore be accessed conveniently, even if an access to `prop` involves a chain of multiple prototype lookups to locate the property. Moreover, objects in JavaScript are passed by reference. Therefore, any operations that modify the object via a reference to it, such

as those in a function to which the object is passed as a parameter, will also modify its label appropriately when the interpreter accesses the in-memory representation of that object.

JavaScript in a browser closely interacts with several browser subsystems. Notably, the browser provides the `document` and `window` interfaces via which JavaScript code can interact with the DOM, *e.g.,* a JSE can access and modify `window.location`. However, such browser objects are not stored and managed by the JavaScript interpreter. Rather, each access to a browser object results in a cross-domain call that gets/sets the value of the browser object. To store security labels for such objects, Sabre also modifies the browser's DOM subsystem to store security labels. Each DOM node has an associated security label. This label is accessed and transmitted by the browser to the JavaScript interpreter when the DOM node is accessed in a JSE.

In addition to the DOM, cross-domain interfaces such as XPCOM allow a JSE to interact with other browser subsystems, such as storage and networking. For example, the following snippet uses XPCOM's cookie manager.

```
1  var cookieMgr =
           Components.classes["@mozilla.org/cookiemanager;1"].
           getService(Components.interfaces.nsICookieManager);
2  var e = cookieMgr.enumerator;
```

In this case, the reference to `enumerator` is resolved via a cross-domain call to the cookie manager. Sabre must separately manage the security labels of `cookieMgr` and those of its properties because `cookieMgr` is not a JavaScript object. Sabre assigns a default security label to cross-domain objects (described in Section 4.4.2). It also ensures that properties that are resolved via cross-domain calls inherit the labels of their parent objects, *e.g.,* `cookieMgr.enumerator` inherits the label of `cookieMgr`.

4.4.2 Sources and Sinks

Sabre detects flows from sensitive sources to low-sensitivity sinks. We consider several sensitive sources, as summarized in Table 4.1, which primarily deal with access to DOM elements, as well as sources enabled by cross-domain access, including those that allow access to persistent storage. Any data received over these interfaces is considered sensitive.

Entity	Sensitive attributes/Method of access
1. Document	cookie, domain, forms, lastModified, links, referrer, title, URL
2. Form	action
3. Form input	checked, defaultChecked, defaultValue, name, selectedIndex, toString, value
4. History	current, next, previous, toString
5. Select option	defaultSelected, selected, text, value
6. Location/Link	hash, host, hostname, href, pathname, port, protocol, search, toString
7. Window	defaultStatus, status
8. Files/Streams	nsIInputStream, nsIFileInputStream, nsILocalFile, nsIFile
9. Passwords	nsIPasswordManager, nsIPasswordManagerInternal
10. Cookies	nsICookieService, nsICookieManager
11. Preferences	nsIPrefService, nsIPrefBranch
12. Bookmarks	nsIRDFDataSource

Table 4.1: **List of sensitive sources in Web browsers.**

Entity	Method of access
1. Files/Processes	nsIOutputStream, nsIFileOutputStream, nsIFile, nsIProcess nsIDownload
2. Network	nsIXMLHttpRequest, nsIHttpChannel, nsITransport
3. DOM	Submission of sensitive DOM node over the network

Table 4.2: **List of low-sensitivity sinks in Web browsers.**

Low-sensitivity sinks accessible from the JavaScript interpreter include the file system and the network, as summarized in Table 4.2. These sinks can further be classified into fine-grained low-sensitivity sinks, such as those based on untrusted domains or file system partitions. In addition to modifying the JavaScript interpreter to raise an alert when a sensitive object is written to a low-sensitivity sink, Sabre also modifies the browser's document interface to raise an alert when a DOM node that stores sensitive data derived from a JSE is sent over the network. For example, Sabre raises an alert when a `form` or a `script` element that contains sensitive data (*i.e.,* data derived from the cookie or password store) is transmitted over the network.

The browser itself may perform several operations that result in information flows from sensitive sources to low-sensitivity sinks. For example, the file system is listed both as a sensitive source and a low-sensitivity sink. This is because a JSE may potentially leak confidential data from a Web application by storing this data on the file system, which may then be accessed by other JSEs or malware on the host machine. However, the browser routinely reads and writes to the file system, *e.g.,* bookmarks and user preferences are read from the file system when the browser starts and are written back to disk when the browser shuts down. To avoid raising an alert on such benign flows, Sabre reports an information flow violation only if an object is written to by a JSE (as discussed in Section 4.4.1). Consequently, it does not report

an alert on benign flows, such as the browser reading and writing user preferences. Even so, a benign JSE may contain instances of flows from sensitive sources to low-sensitivity sinks as part of its advertised behavior. Disallowing such flows may render the JSE dysfunctional. In Section 4.4.4, we discuss how Sabre handles such flows via whitelisting.

While sources and sinks listed above help detect confidentiality-violating information flows, a similar set of low-integrity sources and high-integrity sinks can also be used to detect integrity violations. In this case, Sabre detects information flows from low-integrity sources, *e.g.,* the network, to high-integrity sinks, *e.g.,* calls to `nsIProcess`, which can be used to start a process on the host system.

4.4.3 Propagating Labels

Sabre modifies the interpreter to additionally propagate security labels. JavaScript instructions can roughly be categorized into assignments, function calls, and control structures, such as conditionals and loops.

Explicit Flows

Sabre handles assignments in the standard way by propagating the label of the RHS of an assignment to its LHS. If the RHS is a complex arithmetic/logic operation, the result is considered sensitive if any of the arguments is sensitive. Assignments to complex objects deserve special care because JavaScript supports dynamic creation of new object properties. For example, the assignment `obj.prop = 0` adds a new integer property `prop` to `obj` if it does not already exist. Recall that Sabre associates a separate label with `obj` and `obj.prop` (in contrast to [157]). In this case, the property `prop` inherits the label of `obj` when it is initially created, but the label may change because of further assignments to `prop`. An aggregate operation on the entire object (*e.g.,* a `length` operation on an array) will use the label of the object. In this case, the label of the object is calculated (lazily, when the object is used) to be the aggregate of the labels of its child properties, *i.e.,* an object is considered sensitive if any of its constituent properties stores sensitive information. Sabre handles arrays in a similar fashion by associating each array element with its own security label. However, the label of

the entire array is the aggregate of its members; doing so is important to prevent unintentional information leaks [157].

Sabre handles function calls in a manner akin to prior work [157]. The execution of a function may happen within a *labeled scope* (described below), in which case the labels of variables modified in the function are combined with the label of the current scope. The scope of a function call such as `obj.func()` automatically inherits the label of the parent object `obj`. `eval` statements are handled similar to function calls; all variables modified by code within an `eval` inherit the label of the scope in which the `eval` statement executes.

Cross-domain function calls require special care. For example, consider the following call, which initializes a `nsIScriptableInputStream` object (`sis`) using a `nsIInputStream` object (`is`): `sis.init(is)`. In this statement, `sis` is not a JavaScript object. The function call to `init` is therefore resolved via a cross-domain call. To handle cross-domain calls, we supplied Sabre with a set of *cross-domain function models* that specify how labels must be propagated across such calls. For example, in this case, the model specifies that the label of `is` must propagate to `sis`. We currently use 127 function models that specify how labels must be propagated for cross-domain calls.

Implicit Flows

While the above statements are examples of explicit data dependencies, conditions (and closely related statements, such as loops and exceptions) induce implicit information flows. In particular, there is a control dependency between a conditional expression and the statements executed within the conditional. Thus, for instance, all statements in both the `T` and `F` blocks in the following statement must be considered sensitive, because `document.cookie.length` is a considered sensitive:

```
if (document.cookie.length > 0) then {T} else {F}
```

Sabre handles implicit flows using labeled scopes. Each condition induces a scope for both its true and false branches. The scope of each branch inherits the label of its conditional; scopes also nest in the natural way. All objects modified within each branch inherit the label of the scope in which they are executed.

```
1  x = false; y = false;
2  if (document.cookie.length > 0)
3  then {x = true} else {y = true}
4  if (x == false) {A}; if (y == false) {B}
```

Figure 4.4: **Example of an implicit information flow that cannot be detected using labeled scopes**

While scopes handle a limited class of implicit information flows, it is well-known that they cannot prevent all implicit flows. For instance, consider the example shown in Figure 4.4 (adapted from [42, 157]). In this figure, one of block `A` or `B` executes, depending on the result of the first conditional. Consequently, there is an implicit information flow from `document.cookie.length` to *both* `x` and `y`. However, a dynamic approach that uses scopes will only mark one of `x` or `y` as sensitive, thereby missing the implicit flow.

Precisely detecting such implicit flows requires static analysis. However, we are not aware of static analysis techniques for JavaScript that can detect all such instances of implicit flow. Although prior work [157] has developed heuristics to detect simple instances of implicit flows, such as the one in Figure 4.4, these heuristics fail to detect implicit flows in dynamically generated code, *e.g.,* code executed as the result of an `eval`. Large, real-world JSEs contain several such dynamic code generation constructs. For example, we found several instances of the `eval` construct in about 50% of the JSEs that we used in our evaluation (Section 4.6). Our current prototype of Sabre therefore cannot precisely detect all instances of implicit flows. A future direction could be to investigate a hybrid approach that alternates static and dynamic analysis to soundly detect all instances of implicit flows.

Instruction Provenance

In addition to propagating sensitivity values, Sabre uses the provenance of each JavaScript instruction to determine whether a JavaScript object is modified by a JSE. If so, it sets a Boolean flag (Section 4.4.1) and records the name of the JSE in the security label of the object for diagnostics. Because the JavaScript interpreter can precisely determine the source file containing the bytecode currently being executed, this approach robustly determines the provenance of an instruction, even if it appears in a XUL overlay that is integrated into the browser core.

4.4.4 Declassifying and Endorsing Flows

As discussed in Section 4.4.2, a benign JSE can contain information flows that may potentially be classified as violations of confidentiality or integrity. For example, consider the PwdHash [144] JSE, which customizes passwords to prevent phishing attacks. This JSE reads and modifies a sensitive resource (*i.e.,* a password) from a Web form, which is then transmitted over the network when the user submits the Web form. Sabre raises an alert because an untrusted JSE can use a similar technique to transmit passwords to a remote attacker. However, PwdHash customizes an input password `passwd` to a `domain` by converting it into SHA1(`passwd||domain`), which is then written back to a DOM element whose origin is `domain`. In doing so, PwdHash effectively *declassifies* the sensitive password. Consequently, this information flow can be whitelisted by Sabre.

To support declassification of sensitive information, Sabre extends the JavaScript interpreter with the ability to declassify flows. A security analyst supplies a *declassification policy*, which specifies how the browser must declassify a sensitive object. Flows that violate integrity can similarly be handled with an *endorsement policy*. Sabre supports two kinds of declassification (and endorsement) policies: *sink-specific* and *JSE-specific*. A sink-specific policy permits fine-grained declassification of objects by allowing an analyst to specify the location of a bytecode instruction and the object externalized by that instruction. In turn, the browser reduces the sensitivity of the object when that instruction is executed. For example, the security analyst would specify the file, function and line number at which to execute the declassification bytecode on the object being externalised. In case of PwdHash, the policy would be the tuple `<stanford-pwdhash.js, finish, 330, field.value>`. In contrast, a JSE-specific policy permits declassification of *all* flows from a JSE and can be used when a JSE is trusted.

Declassification (and endorsement) policies must be supplied with care because declassification causes Sabre to allow potentially unsafe flows. In the experiments reported in Section 4.6, we manually wrote declassification policies by examining execution traces emitted by Sabre and determining whether the offending flow is part of the advertised behavior of the JSE. If the flow was advertised by the JSE, we wrote a sink-specific policy to allow that flow.

4.5 Implementation

We implemented Sabre by modifying SpiderMonkey, the JavaScript interpreter in Firefox, to track information flow. We chose Firefox as our implementation and evaluation platform because of the popularity and wide availability of JSEs for Firefox. However, JSEs pose a security threat even in privilege-separated browser architectures (*e.g.,* [10,83,162]) for the same reasons as outlined earlier in Section 4.1. The techniques described here are therefore relevant and applicable to such browsers as well.

We modified SpiderMonkey's representation of JavaScript objects to include security labels. We also enhanced the interpretation of JavaScript bytecode instructions to modify labels, thereby propagating information flow. We also modified other browser subsystems, including the DOM subsystem (*e.g.,* HTML, XUL and SVG elements) and XPCOM, to store and propagate security labels, thereby allowing information flow tracking across browser subsystems. This approach allows us to satisfy our design goals. All JavaScript code is executed by the interpreter, thereby ensuring complete mediation even in the face of browser vulnerabilities, such as those discussed in Section 4.1. Moreover, associating security labels directly with JavaScript objects and tracking these labels within the interpreter and other browser subsystems makes our approach robust to obfuscated JavaScript code, *e.g.,* as may be found in drive-by-download Web sites that attempt to exploit browser and JSE vulnerabilities. Finally, the interpreter can readily identify the source of the JavaScript bytecode currently being interpreted, thereby allowing for easy action attribution.

Although Sabre's approach of using browser modifications to ensure JSE security is not as readily portable as, say, language restrictions [49,72,119], this approach also ensures compatibility with legacy JSEs. For example, Adsafe [49] would reject JSEs containing dynamic code generation constructs, such as `eval`; in contrast, Sabre allows arbitrary code in a JSE, but instead tracks information flow. An information flow tracker based on JavaScript instrumentation using the recently standardized `Proxy` objects [8] will be portable across browsers.

4.6 Evaluation

We evaluated Sabre using a suite of 24 JSEs, comprising over 120K lines of JavaScript code. Our goals were to test both the effectiveness of Sabre at analyzing information flows and to evaluate its runtime overhead.

4.6.1 Effectiveness

Our test suite included both JSEs with known instances of malicious flows as well as those with unknown flows. In the latter case, we used Sabre to understand the flows and determine whether they were potentially malicious.

• **JSEs with known malicious flows.** We evaluated Sabre with four JSEs that had known instances of malicious flows. These included two JSEs that contained exploitable vulnerabilities (Greasemonkey v0.3.3 and Firebug v1.01) and two publicly-available malicious JSEs (FFSniFF [25] and BrowserSpy [104]).

To test vulnerable JSEs, we adapted information available in public fora [24, 136, 146, 166] to write Web pages containing malicious scripts. The exploit against Greasemonkey attempted to transmit the contents of a file on the host to an attacker, thereby violating confidentiality, while exploits against Firebug attempted to start a process on the host and modify the contents of a file on disk, thereby violating integrity. In each case, Sabre precisely identified the information flow violation. We also confirmed that Sabre did not raise an alert when we used a JSE-enhanced browser to visit benign Web pages.

To test malicious JSEs, we considered FFSniFF and BrowserSpy, both of which exhibit the same behavior—they steal passwords and other sensitive entries from Web forms and hide their presence from the user by removing themselves from the browser's extension manager. Nevertheless, because Sabre records the provenance of each JavaScript bytecode instruction executed, it raised an alert when FFSniFF and BrowserSpy attempted to transmit passwords to a remote attacker via the network.

In addition to the above JSEs, we also wrote a number of malicious JSEs, both to demonstrate the ease with which malicious JSEs can be written and to evaluate Sabre's ability to detect them. Each of our JSEs comprised under 100 lines of JavaScript code, and were written

by an undergraduate student with only a rudimentary knowledge of JavaScript. For example, *ReadCookie* is a JSE that reads browser cookies and stores them in memory. When the user visits a particular Web page (in our prototype, any Web page containing Google's search utility), the JSE creates a hidden form element, stores the cookies on this form, and modifies the action attribute to redirect the search query to a malicious server address. The server receives both the search query as well as the stolen cookies via the hidden form element. Sabre detects this malicious flow when the user submits the search request because the hidden form field that stores cookies (and is therefore labeled sensitive) is transmitted over the network.

• **JSEs with unknown information flows.** In addition to testing Sabre against known instances of malicious flows, we tested Sabre against 20 popular Firefox JSEs. The goal of this experiment was to understand the nature of information flows in these JSEs and identify suspicious flows.

Our experimental methodology was to enhance the browser with the JSE being tested and examine any violations reported by Sabre. We would then determine whether the violation was because of advertised functionality of the JSE, in which case we whitelisted the flow using a sink-specific declassification or endorsement policy, or whether the flow was indeed malicious. Although we ended up whitelisting suspicious flows for all 20 JSEs, our results described below show that information flows in several of these JSEs closely resemble those exhibited by malicious extensions, thereby motivating the need for a fine-grained approach to certify information flows in JSEs.

In our experiments, which are summarized in Table 4.3, we found that the behavior of JSEs in our test suite fell into five categories. As Table 4.3 illustrates, several JSEs contained a combination of the following behaviors.

1. **Interaction with HTML forms.** An HTML form is a collection of form elements that allows users to submit information to a particular domain. Example of form elements include login names, passwords and search queries. While malicious JSEs (*e.g.,* FFsniFF) can steal data by reading form elements, we also found that PwdHash [144] reads information from form elements.

	JSE	Advertised Functionality of JSE	1	2	3	4	5
1.	Adblock Plus	Prevents download of page elements, such as ads		✓	✓		
2.	All-in-One-Sidebar	Enables switch b/w sidebar panels & dialog windows			✓		
3.	CoolPreviews	Preview links & images without leaving current page		✓	✓		
4.	Download Statusbar	Manage downloads from a tidy statusbar			✓		
5.	Fast Video Download	Easy download of video files from popular sites				✓	
6.	Forecastfox	Gets weather forecasts from AccuWeather.com		✓	✓	✓	
7.	Foxmarks Synchronizer	Synchronizes and backs up bookmarks and passwords		✓	✓		
8.	Ghostery	Alerts users about Web bugs and trackers on Web pages			✓		
9.	GooglePreview	Adds thumbnails & site ranks in Google search results		✓	✓		
10.	Greasemonkey (0.8.1)	Allows customization of Web pages with user scripts		✓	✓		
11.	NoScript	Restricts executable content to trusted domains		✓	✓		
12.	PDF Download	Tool for viewing and creating Web-based PDF files		✓	✓	✓	
13.	Pwdhash	Customizes passwords to domains to prevent phishing	✓				
14.	SpeedDial	Easy access to frequently visited Web sites			✓	✓	
15.	StumbleUpon	Discovers Web sites based on user's interests		✓	✓	✓	
16.	Stylish	Enhances browsing experience by managing user styles		✓	✓	✓	✓
17.	Tab Mix Plus	Enhances Firefox's tab browsing capabilities			✓	✓	
18.	User Agent Switcher	Switches the user agent of the browser			✓		
19.	Video DownloadHelper	Tool for Web content extraction		✓	✓		
20.	Web-of-Trust	Warns users before they interact with a harmful site		✓	✓	✓	

Behavior key: **(1)** HTML forms; **(2)** HTTP channels; **(3)** File system; **(4)** Loading URLS; **(5)** JavaScript events.

Table 4.3: **Behavior of popular Firefox JSEs analyzed using Sabre.** The extension behavior is further categorized as in Section 4.6.1

PwdHash recognizes passwords prefixed with a special symbol ("`@@`") and customizes them to individual domains to prevent phishing attacks. In particular, it reads the password from the HTML form, transforms it as described in Section 4.4.4, and writes the password back to the HTML form. This behavior can potentially be misused by an untrusted JSE, *e.g.,* a malicious JSE could read and maliciously modify form elements when the user visits a banking Web site, thereby compromising integrity of banking transactions. Consequently, Sabre marks the HTML form element containing the password as sensitive, and raises an alert when the form is submitted. However, because the information flow in PwdHash is benign, we declassify the customized password before it is written back to the form, thereby preventing Sabre from raising an alert.

2. **Sending/receiving data over an HTTP channel.** JSEs extensively use HTTP messages to send and receive data, either via `XMLHttpRequest` or via HTTP channels. For example, Web-of-Trust is a JSE that performs an `XMLHttpRequest` for each URL that a user visits, in order to fetch security ratings for that URL from its server.

   While this behavior can potentially be misused by malicious JSEs to compromise user privacy by exposing the user's surfing patterns, we allowed the `XMLHttpRequest` in Web-of-Trust by declassifying the request.

3. **Interaction with the file system.** With the exception of two JSEs, the rest of the JSEs in our test suite interacted with the file system. For example, Video DownloadHelper and Greasemonkey download content from the network on to the file system (media files and user scripts, respectively), while ForecastFox reads user preferences, such as zip codes, from the file system and sends an `XMLHttpRequest` to receive weather updates from accuweather.com.

   Both these behaviors can potentially be misused by malicious JSEs to download malicious files on the host and steal confidential data, such as user preferences. However, we allowed these flows by endorsing the file system write operation in Video DownloadHelper and Greasemonkey and by declassifying the `XMLHttpRequest` in ForecastFox.

4. **Loading a URL.** Several JSEs, such as SpeedDial and PDF Download, monitor user

activity (*e.g.,* keystrokes, hyperlinks clicked by the user) and load a URL based upon this activity. For example, PDF Download, which converts PDF documents to HTML files, captures user clicks on hyperlinks and sends an `XMLHttpRequest` to its home server to get a URL to a mirror site. It then constructs a new URL by appending the mirror's URL with the hyperlink visited by the user, and loads the newly-construced URL in a new tab.

Similar behavior can potentially be misused by a JSE, *e.g.,* to initiate a drive-by-download attack by loading an untrusted URL. However, for PDF Download, we endorsed the JavaScript statements that load URLs in the JSEs that we tested, thereby preventing Sabre from raising an alert.

5. **JavaScript events.** Unprivileged JavaScript code on a Web page can communicate with privileged JavaScript code (*e.g.,* code in JSEs) via events. In particular, JSEs can listen for specific events from scripts on Web pages.

   We found one instance of such communication in the Stylish JSE, which allows easy management of CSS styles for Web sites. A user can request a new style for a Web page, in response to which the JSE opens a new tab with links to various CSS styles. When the user chooses a style, JavaScript code on Web page retrieves the corresponding CSS style and throws an event indicating that the download is complete. Stylish captures this event, extracts the CSS code, and opens a dialog box for the user to save the file.

   Sabre raises an alert when the user saves the file. This is because Sabre assigns a low integrity label to JavaScript code on a Web page; in turn the event thrown by the code also receives this label. Sabre reports an integrity violation, when the JavaScript code in Stylish handles the low-integrity event and attempts to save data on to the file system (a high-integrity sink). Nevertheless, we suppressed the alert by endorsing this flow.

Sabre provides detailed traces of JavaScript execution for offline analysis. We used these traces in our analysis of JSEs to determine whether an information flow was benign, and if so, determine the bytecode instruction and the JavaScript object at which to execute the declassification/endorsement policy. Although this analysis is manual, in our experience, it only took on

the order of a few minutes to determine where to place declassifiers.

As the examples above indicate, several benign JSEs exhibit information flows that can possibly be misused and must therefore be analyzed and whitelisted. It is important to note that each of these information flows exhibited *real* behaviors in JSEs. Because such behaviors may possibly be misused by malicious JSEs, determining whether to whitelist a flow is *necessarily* a manual procedure, *e.g.,* of studying the high-level specification of the JSE to determine if the behavior conforms to the specification.

To evaluate the precision of Sabre, we also studied whether it reported any *other* instances of flows from sensitive sources to low-sensitivity sinks, *i.e.,* excluding the flows that were whitelisted above. We used a Sabre-enhanced browser for normal Web browsing activity over a period of several weeks. During this period, Sabre, reported *no* violations. We found that Sabre's policy of reporting an information flow violation only when an object is modified by a JSE was crucial to the precision of Sabre.

The analysis above shows that benign JSEs often contain information flows that can potentially be misused by malicious JSEs. These results therefore motivate a security architecture for JSEs in which JSE vendors explicitly state information flows in a JSE by supplying a declassification/endorsement policy for confidentiality/integrity violating flows. This policy must be approved by the user (or a trusted third party, such as `addons.mozilla.org`, that publishes JSEs) when the JSE is initially installed and is then enforced by the browser.

It is important to note that this architecture is agnostic to the *code* of a JSE and only requires the user to approve *information flows*. In particular, the declassification policy is decoupled from the code of the JSE is enforced by the browser. As a result, only flows whitelisted by the user will be permitted by the browser, thereby significantly constraining confidentiality and integrity violations via JSEs. This architecture also has the key advantage of being robust even in the face of attacks enabled by vulnerabilities in the JSE.

4.6.2 Performance

We evaluated the performance of Sabre by integrating it with SpiderMonkey in Firefox 2.0.0.9. Our test platform was a 2.33Ghz Intel Core2 Duo machine running Ubuntu 7.10 with 3GB

RAM. We used the SunSpider [20] and V8 [81] JavaScript benchmark suites to evaluate the performance of Sabre. Our measurements were averaged over ten runs.

With the V8 suite, a Sabre-enabled browser reported a mean score of 29.16 versus 97.91 for an unmodified browser, an overhead of 2.36×, while with SunSpider, a Sabre-enabled browser had an overhead of 6.1×. We found that the higher overhead in SunSpider was because of three benchmarks (3d-morph, access-nsieve and bitops-nsieve-bits). Discounting these three benchmarks, Sabre's overhead with SunSpider was 1.6×. Despite these overheads, the performance of the browser was not noticeably slower during normal Web browsing, even with JavaScript-heavy Web pages, such as Google Maps and Street View.

The main reason for the high runtime overhead reported above is that Sabre monitors the provenance of *each* JavaScript bytecode instruction to determine whether the instruction is from a JSE (to set the Boolean flag in the security label, as described in Section 4.4.3). Monitoring each instruction is important primarily because code included in overlays (distributed with JSEs) is included in the browser core and may be executed at any time. If such overlays can separately be verified to be benign, these checks can be disabled. In particular, when we disabled this check, we observed a manageable overhead of 77% and 42% with the V8 and SunSpider suites, respectively. Ongoing efforts by Eich *et al.* [65,66] to track information flow in JavaScript also incur comparable (20%-70%) overheads.

4.7 Related Work

While early work by Hallaraker and Vigna [89] proposed XPCOM-level monitoring to sandbox JavaScript code, Ter-Louw *et al.* [104] were the first to really address the security implications of JSEs. However, their work was based on monitoring XPCOM calls and was coarse-grained. Their approach can have both false positives and negatives.

Sabre is most closely related to [59] by Djeric and Goel, which, like Sabre, presents a dynamic information flow tracking system to detect insecure flow patterns in JavaScript extensions with the goal of detecting JavaScript-based extensions that can leak or misuse sensitive browser data. Unlike Sabre, VEX [33, 34] implements a static analysis of JavaScript to study

vulnerabilities in JavaScript-based Web browser extensions. It implements a flow- and context-sensitive analysis that was applied to over 2400 JavaScript-based Firefox extensions to detect unsafe programming practices. In VEX, vulnerabilities are specified as bad flow patterns; the analysis attempts to verify the absence of these patterns in JavaScript-based browser extensions.

The author has also contributed to the development of Beacon—details of this tool are available in [96]. Unlike Sabre, Beacon implements static information flow for JavaScript, and was applied to Mozilla's Jetpack framework and over 350 Jetpack extensions. It detected several critical capability leaks and violations in principle of least authority [18], even in production quality code by Mozilla.

IBEX [87] is a framework for specifying fine-grained access control policies guarding the behavior of monolithic browser extensions. Like Sabre and VEX, IBEX is also a tool for extension curators to detect policy violating JavaScript extensions. But, there are several differences as well. While IBEX uses access control techniques, Sabre performs information flow for JavaScript extensions and is designed to detect confidentiality and integrity violations. IBEX also requires extensions to first be written in a dependently-typed language (to make them amenable to verification), following which they are translated to JavaScript. In contrast, Sabre works directly with legacy extensions written in JavaScript.

New extension frameworks, like Mozilla's Jetpack [16] and Google Chrome's extension architecture [35], encourage a modular extension design. Such extensions can consist of both scriptable and native components and require each extension to specify its resource requirements upfront in a manifest. The contents of the manifest are then enforced by the extension framework, thereby limiting the effect of any exploits against the extension. However, recent works have shown that this model might be insufficient to ensure the security of both the Jetpack extensions [96] and Chrome extensions [41, 100].

FlowFox [53] implements a new dynamic enforcement mechanism within the Web browser for information flow security. This technique, also called secure multi-execution [55], executes the program multiple times – once for every security label. Although different in its implementation, FlowFox is similar in spirit to Sabre and [157], both of which use JavaScript-level taint tracking.

Lastly, systems like Xax [61] and NaCl [175], have explored techniques to sandbox browser extensions, but such work is currently applicable only to extensions such as plugins and BHOs, which are distributed as binary executables. Contrary to such techniques, Sabre works for JavaScript-based browser extensions. Cavallaro *et al.* [42] developed several techniques that malicious software could use to defeat information flow trackers. Among the attacks presented in that paper, Sabre is vulnerable to JSEs that use certain forms of implicit information flows, as discussed in Section 4.4.3.

4.8 Summary

In this chapter, we present Sabre, which implements JavaScript-level information flow analysis to study and characterize the behavior of JavaScript-based Web browser extensions. We use Sabre to report on extensive experiments about the nature of vulnerabilities in legacy Web browser extensions. Sabre's findings and our experience with Beacon suggest that legacy extension platforms are vulnerable to a large class of attacks, and even the more modern extension frameworks are not yet fully secure.

Chapter 5

Language-based Security for Web Platform Extensions

In the previous chapter, we have studied and characterized the behavior of untrusted, third party JavaScript-based Web browser extensions. We now address the issue of such insecure JavaScript-based Web browser and Web application extensions using Transcript, a language runtime system, which implements *isolation* as a first class primitive for JavaScript. Transcript also enhances the JavaScript language with builtin support for introspection of untrusted third party JavaScript code.

In this chapter, we formalize Transcript, discuss its design and implementation in a commodity Web browser, and evaluate its effectiveness on real-world Web applications. Although, Transcript focuses on Web application extensions, like advertisements, widgets and third party JavaScript libraries, the techniques presented here are equally applicable to Web browser extensions.

5.1 Problem

It is now common for Web applications (*host*) to include untrusted third party JavaScript code (*guest*) of arbitrary provenance, in the form of advertisements, libraries and widgets. Despite advances in language and browser technology, JavaScript still lacks mechanisms that enable Web application developers to debug and understand the behavior of third party guest code. Using existing reflection techniques in JavaScript, the host cannot attribute changes in the JavaScript heap and the DOM to specific guests. Further, fine-grained context about guest's interaction with host's DOM and network is not supported. For example, the host cannot inspect the behavior of guest code under specific cookie values or decide whether to allow network requests by the guests.

```
1  <script type="text/javascript">
2  var editor = new Editor(); initialize(editor);
3  var builtins = [], tocommit = true;
4  for(var prop in Editor.prototype) builtins[prop] = prop;
5  var tx = transaction {
     // Guest code:  Lines 6--9
6    Editor.prototype.getKeywords = function(content) {...}
     ...
7    var elem = document.getElementById("editBox");
8    elem.addEventListener("mouseover", displayAds, false);
     ...
9    document.write('<div style="opacity:0.0; z-index:0;
       ...  size/loc params"> <a href="http://evil.com">
        Evil Link </a></div>');
10 };
   // gotoIblock implements the host's security policies
11 tocommit = gotoIblock(tx);
12 if (tocommit) tx.commit();
13 ...  /* rest of the Host Web application's code */
14 </script>
```

Figure 5.1: **A motivating example for Transcript.** This example shows how a host can mediate an untrusted guest (lines 6–9). The introspection block (invoked in line 11) enforces the host's security policies (see Figure 5.2) on the actions performed by the guest.

5.2 Motivating Example

Let us consider an example of a Web-based word processor that hosts a third party widget to display advertisements (see Figure 5.1). During an editing session, this widget scans the document for specific keywords and displays advertisements relevant to the text that the user has entered. Such a widget may modify the host in several ways to achieve its functionality, *e.g.,* it could install event handlers to display advertisements when the user places the mouse over specific phrases in the text. However, as an untrusted guest, this widget may also contain malicious functionality, *e.g.,* it could implement a clickjacking-style attack by overlaying the editor with transparent HTML elements pointing to malicious sites.

Traditional reference monitors [69], which mediate the action of guest code as it executes, can detect and prevent such attacks. However, such reference monitors typically only enforce access control policies, and would have let the guest modify the host's heap and DOM (such as to install innocuous event handlers) until the attack is detected. When such a reference monitor reports an attack, the end-user faces one of two unpalatable options: (a) close the editing session and start afresh; or (b) continue with the tainted editing session. In the former case, the end-user

loses unsaved work. In the latter case, the editing session is subject to the unknown and possibly undesirable effects of the heap and DOM changes that the widget initiated before being flagged as malicious. In our example, the event handlers registered by the malicious widget may also implement undesirable functionality and should be removed when the widget's clickjacking attempt is detected.

5.3 Our Approach: Speculative Execution of JavaScript

The main idea is to extend JavaScript with a new `transaction` construct, within which hosts can speculatively execute guest code containing arbitrary JavaScript constructs. In addition to enforcing security policies on guests, a `transaction` would allow hosts to *cleanly recover from policy-violating actions of guest code*. When a host detects an offending guest, it simply chooses not to commit the transaction corresponding to the guest. Such an approach neutralizes any data and DOM modifications initiated earlier by the guest without having to undo them explicitly. The introspection mechanism (`transaction`) is built within the JavaScript language itself, thereby allowing guest code to contain arbitrary JavaScript constructs (unlike contemporary techniques [49, 72, 107, 109, 119]).

Speculative execution allows hosts to introspect all actions of untrusted guest code. In our example, the host speculatively executes the untrusted widget by enclosing it in a transaction. When the attack is detected, the host simply discards all changes initiated by the widget. The end-user can proceed with the editing session without losing unsaved work, and with the assurance that the host is unaffected by the malicious widget.

This chapter describes the Transcript system, which has the following novel features:

- **JavaScript transactions.** Transcript allows hosting Web applications to speculatively execute guests by enclosing them in transactions. Transcript maintains *read and write sets* for each transaction to record the objects that are accessed and modified by the corresponding guest. These sets are exposed as properties of a *transaction object* in JavaScript. Changes to a JavaScript object made by the guest are visible within the transaction, but any accesses to that object from code outside the transaction return the unmodified object. The host can inspect such speculative changes made by the guest and

determine whether they conform to its security policies. The host must explicitly commit these changes in order for them to take effect; uncommitted changes simply do not take and need not be undone explicitly.

- **Transaction suspend/resume.** Guest code may attempt operations outside the purview of the JavaScript interpreter. In a browser, these *external operations* include AJAX calls that send network requests, such as `XMLHttpRequest`. Transcript introduces a *suspend and resume* mechanism that affords flexibility to mediate external operations. Whenever a guest attempts an external operation, Transcript suspends it and passes control to the host. Depending on its security policy, the host can perform the action on behalf of the guest, perform a different action unbeknownst to the guest, or buffer up and simulate the action, before resuming this or another suspended transaction.

- **Speculative DOM updates.** Because JavaScript interacts heavily with the DOM, Transcript provides a speculative DOM subsystem, which ensures that DOM changes requested by a guest will also be speculative. Together with transactions, Transcript's DOM subsystem allows hosts to cleanly recover from attacks by malicious guests.

Transcript provides these features without restricting or modifying guest code in any way. This allows reference monitors based on Transcript to mediate the actions of legacy libraries and applications that contain constructs that are often disallowed in safe JavaScript subsets [49, 72, 107, 109, 119] (*e.g.,* `eval`, `this` and `with`).

5.4 Overview of Transcript

Transcript enables hosts to understand the behavior of untrusted guests, detect attacks by malicious guests and recover from them, and perform forensic analysis. We briefly discuss Transcript's utility and then provide an overview of its functionality for confining a malicious guest.

- **Understanding guest code.** Analysis of third party JavaScript code is often hard, due code obfuscation. Using Transcript, a host can set watchpoints on objects of interest. Coupled with suspend/resume, it is possible to perform a fine grained debug analysis by inspecting the read/write sets on every guest initiated object read/write and method

invocation. Transcript's speculative execution provides an ideal platform for concolic unit testing [79, 149] of guests. For example, using Transcript, a host can test a guest's behavior under different values of domain cookies.

- **Confining malicious guests.** Transcript's speculative execution permits buffering of network I/O and writing to a speculative DOM, thereby allowing flexibility in confining untrusted guest code. For example, to prevent clickjacking-style attacks, the host can simply discard guest's modifications to the speculative DOM.

- **Forensic analysis.** Since Transcript suspends on external and user-defined operations, the suspend/resume mechanism is an effective tool for forensic analysis of a suspected vulnerability exploited by the guest. For example, code-injection attacks using DOM or host APIs [7] can be analyzed by observing the sequence of suspend calls and their arguments.

Transcript in Action

We illustrate Transcript's ability to confine untrusted guests by further elaborating on the example introduced in Section 5.2. Suppose that the word processor hosts the untrusted widget using a `<script>` tag, as follows: `<script src="http://untrusted.com/guest.js">`. In Figure 5.1, lines **6–9** show a snippet from `guest.js` that displays advertisements relevant to keywords entered in the editor. Line **6** registers a function to scan for keywords in the editor window by adding it to the prototype of the `Editor` object. Lines **7** and **8** show the widget registering an event handler to display advertisements on certain mouse events. While lines **6–8** encode the core functionality related to displaying advertisements, line **9** implements a clickjacking-style attack by creating a transparent `<div>` element, placed suitably on the editor with a link to an evil URL.

When hosting such a guest, the word processor can protect itself from attacks by defining and enforcing a suitable set of security policies. These may include policies to prevent prototype hijacks [132], clickjacking-style attacks, drive-by downloads, stealing cookies, snooping on keystrokes, *etc*. Further, if an attack is detected and prevented, it should not adversely affect normal operation of the word processor. We now illustrate how the word processor can use

```
A  do { // host's introspection block:  Lines A--T
B    var arg = tx.getArgs(), obj = tx.getObject();
C    var rs = tx.getReadSet(), ws = tx.getWriteSet();
D    for(var i in builtins) {
E      if (ws.checkMembership(Editor.prototype, builtins[i]))
F        tocommit = false;
G    } ...  /* definition of 'IsClickJacked' to go here */
H    if (IsClickJacked(tx.getTxDocument()))
I       tocommit = false;
J    ...  /* more policy checks go here */
     // inlined code from libTranscript:  Lines K--Q
K    switch(tx.getCause()) {
L      case "addEventListener":
M        var txHandler = MakeTxHandler(arg[1]);
N        obj.addEventListener(arg[0], txHandler, arg[2]); break;
O      case "write":  WriteToTxDOM(obj, arg[0]); break;
       ...  /* more cases */
P      default:  break;
Q    };
R    tx = tx.resume();
S  } while(tx.isSuspended());
T  return tocommit;
```

Figure 5.2: **Example of an application defined introspection block (iblock) to mediate actions of untrusted JavaScript code.** An iblock consists of two parts: a host-specific part, which encodes the host's policies to confine the guest (lines D–J), and a mandatory part, which contains functionality that is generic to all hosts (lines K–Q).

Transcript to achieve such protection and cleanly recover from attempted attacks.

The host protects itself by embedding the guest within a `transaction` construct (line **5**, Figure 5.1) and specifies its security policy (lines **D**–**Q**, Figure 5.2). When the transaction executes, Transcript records all reads and writes to JavaScript objects in per-transaction read/write sets. Any attempts by the guest to modify the host's JavaScript objects (*e.g.,* on line **6**, Figure 5.1) are speculative; *i.e.,* these changes are visible only to the guest itself and do not modify the host's view of the JavaScript heap. To ensure that DOM modifications by the guest are also speculative, Transcript's DOM subsystem clones the host's DOM at the start of the transaction and resolves all references to DOM objects in a transaction to the cloned DOM. Thus, references to `document` within the guest resolve to the cloned DOM.

When the guest performs DOM operations, such as those on lines **7**–**9**, and other external

operations, such as `XMLHttpRequest`, Transcript *suspends* the transaction and passes control to the host. This situation is akin to a system call in a user-space program causing a trap into the operating system. Suspension allows hosts to mediate external operations as soon as the guest attempts them. When a transaction suspends or completes execution, Transcript creates a *transaction object* in JavaScript to denote the completed or suspended transaction. In Figure 5.1, the variable `tx` refers to the transaction object. Transcript then passes control to the host at the program point that syntactically follows the transaction. There, the host implements an *introspection block* (or *iblock*) to enforce its security policy and perform operations on behalf of a suspended transaction.

Transaction Objects

A transaction object records the state of a suspended or completed transaction. It stores the read and write sets of the transaction and the list of activation records on the call stack of the transaction when it was suspended. It provides builtin methods, such as `getReadSet` and `getWriteSet` shown in Figure 5.2, which the host can invoke to access read and write sets, observe the actions of the guest, and make policy decisions.

When a guest tries to perform an external operation and thus suspends, the resulting transaction object contains arguments passed to the operation. For example, a transaction that suspends due to an attempt to modify the DOM, such as the call `document.write` on line **9**, will contain the DOM object referenced in the operation (`document`), the name of the method that caused the suspension (`write`), and the arguments passed to the method recall that Transcript's DOM subsystem ensures that `document` referenced within the transaction will point to the cloned DOM). The host can access these arguments using builtin methods of the transaction object, such as `getArgs`, `getObject` and `getCause`.

Depending on its policy, the host can either perform the operation on behalf of the guest, simulate the effect of performing it, defer the operation for later, or not perform it at all.

The host can resume a suspended transaction using the transaction object's builtin `resume` method. Transcript then uses the activation records stored in the transaction object to restore the call stack, and resumes control at the program point following the instruction that caused

the transaction to suspend (akin to resumption of program execution following a system call). Transactions can suspend an arbitrary number of times until they complete execution. The builtin `isSuspended` method determines whether the transaction is suspended or has completed.

A completed transaction can be committed using the builtin `commit` method. This method copies the contents of the write set to the corresponding objects on the host's heap, thereby publishing the changes made by the guest. It also synchronizes the host's DOM with the cloned version that contains any DOM modifications made by the guest. A completed transaction's call stack is empty, so attempts to resume a completed transaction will have no effect. Note that Transcript does not define an explicit `abort` operation. This is because the host can simply discard changes made by a transaction by choosing not to commit them. If the transaction object is not referenced anymore, it will be garbage-collected.

Introspection Blocks

When a transaction suspends or completes, Transcript passes control to the instruction that syntactically follows the transaction in the code of the host. At this point, the host can check the guest's actions by encoding its security policies in an *iblock*. The iblock in Figure 5.2 spans lines **A**–**T** and has two logical parts: a *host-specific part*, which encodes host's policies (lines **D**–**J**), and a *mandatory part*, which performs operations on behalf of suspended guests (lines **K**–**Q**). The iblock in Figure 5.2 illustrates two policies:

- Lines **D**–**F** detect prototype hijacking attempts on the `Editor` object. To do so, they check the transaction's write set for attempted redefinitions of builtin methods and fields of the `Editor` object.

- Lines **H**–**I** detect clickjacking-style attempts by checking the DOM for the presence of any transparent HTML elements introduced by the guest. The body of `IsClickJacked`, which implements the check, is omitted for brevity.

The body of the `switch` statement encodes the mandatory part of the iblock and implements two key functionalities, which are further explained in Section 5.6.1:

- Lines **L–N** in Figure 5.2 create and attach an event handler to the cloned DOM when the guest suspends on line **8** in Figure 5.1. The `MakeTxHandler` function creates a new *wrapped* handler, by enclosing the guest's event handler (`displayAds`) within a `transaction` construct. Doing so ensures that the execution of any event handlers registered by the guest is also speculative, and mediated by the host's security policies. The iblock then attaches the event handler to the corresponding element (`elem`) in the cloned DOM.

- Line **O** in Figure 5.2 speculatively executes the DOM modifications requested when the guest suspends on line **9** in Figure 5.1. The `WriteToTxDOM` function invokes the `write` call on `obj`, which points to the `document` object in the cloned DOM.

If a transaction does not commit because of a policy violation, the host's DOM and JavaScript objects will remain unaffected by the guest's modifications. For instance, when the host in Figure 5.1 aborts the guest after it detects the clickjacking attempt, the host's DOM will not contain any remnants of the guest's actions (such as event handlers registered by the guest). The host's JavaScript objects, such as `Editor`, are also unaffected. Speculatively executing guests therefore allows hosts to cleanly recover from attack attempts.

Iblocks offer hosts the option to postpone external operations. For example, a host may wish to defer all network requests from an untrusted advertisement until the end of the transaction. It can do so using an iblock that buffers these requests when they suspend, and thereafter resume the transaction; the buffered requests can be processed after the transaction has completed. Such postponement will not affect the guest if the buffered requests are asynchronous, *e.g.,* asynchronous `XMLHttpRequest`.

Because a transaction may suspend several times, the iblock is structured as a loop whose body executes each time the transaction suspends and once when the transaction completes. This way, the same policy checks apply whether the transaction suspended or completed.

$$
\begin{array}{lll}
\text{Expressions} & M ::= & n \mid \ell \mid x \mid \lambda x.\, M \mid M{+}M \mid MM \mid RW\{M\} \\
& & \mid \mathsf{commit}\, M \mid \mathsf{introspect}\, M(x.M)(x.M) \\
& & \mid \mathsf{new}\, M \mid \mathsf{read}\, M \mid \mathsf{write}\, MM \\
& & \mid \mathsf{suspend}\, M \mid \mathsf{resume}\, MM \\
\text{Values} & V ::= & n \mid \ell \mid \lambda x.\, M \\
& & \mid RW\{V\} \mid RW\{C[\mathsf{suspend}\, V]\} \\
\text{Contexts} & C ::= & \square \mid C{+}M \mid V{+}C \mid CM \mid VC \\
& & \mid \mathsf{commit}\, C \mid \mathsf{introspect}\, C(x.M)(x.M) \\
& & \mid \mathsf{new}\, C \mid \mathsf{read}\, C \mid \mathsf{write}\, CM \mid \mathsf{write}\, VC \\
& & \mid \mathsf{suspend}\, C \mid \mathsf{resume}\, CM \mid \mathsf{resume}\, VC \\
\text{Metacontexts} & D ::= & \square \mid D[RW\{C\}]
\end{array}
$$

Figure 5.3: **Syntax of the core language describing transactions.**

5.5 A Lambda Calculus with Transactions

To explain concisely and formally how transactions underpin the motivating example above, we present a call-by-value lambda calculus with transactions and specify its operational semantics [73]. The essential idea is to use the evaluation context to delimit transactions and isolate them from external resources [98].

5.5.1 Formalization

The syntax of our core language is defined by the grammar in Figure 5.3. A value V is a special case of an expression M. Here, we assume integer constants n (for illustration), an infinite supply of heap locations ℓ, and lexically scoped variables x.

A read/write set RW consists of the read set R and the write set W. Whereas R is a relation between locations and values,[1] W is a partial function from locations to values. For example, suppose that the global heap comprises three locations ℓ_1, ℓ_2, ℓ_3, containing $10, 20, 30$ respectively. The global write set is then $\{\ell_1 \mapsto 10,\ \ell_2 \mapsto 20,\ \ell_3 \mapsto 30\}$. Suppose now a transaction reads 10 from ℓ_1, writes 25 to ℓ_2, reads 30 from ℓ_3, writes 35 to ℓ_3, reads the new value 25 back from ℓ_2, and initializes a new location ℓ_4 to 45. Then, the global write set stays the same, but the read set of the transaction changes from the empty set to $\{\ell_1 \mapsto 10,$

[1] R is a relation, not necessarily a partial function. It may relate one location to multiple values if, while the transaction is suspended, the location is mutated outside, and the transaction reads both the old value before suspending and the new value after resuming.

$\ell_3 \mapsto 30\}$, and the write set of the transaction changes from the empty set to $\{\ell_2 \mapsto 25$, $\ell_3 \mapsto 35$, $\ell_4 \mapsto 45\}$.[2] Read/write sets thus subsume mutable state.

A context C is a special case of an expression in which a subexpression next to be evaluated is replaced by a hole $\Box$. Roughly speaking, whereas a read/write set represents the heap state of an ongoing transaction (akin to the contents of private pages in the address space of a thread), a context represents the control state of an ongoing transaction (akin to the sequence of activation frames on the execution stack of a thread). Whereas many operational semantics (including Maffeis *et al.*'s for JavaScript [106]) leave control state implicit in contextual or congruence rules, we make it explicit so as to specify how transactions suspend. We write $C[M]$ for the expression obtained by replacing the hole in C with M. For example, if $C = \Box 0$ then $C[\lambda x.\, x] = (\lambda x.\, x)0$.

A transaction expression $RW\{M\}$ is formed by *delimiting* an (untrusted) expression M with a read/write set RW (initially empty). This formation is similar to how a try-expression is formed, in a typical language with exception handling, by delimiting an expression with a handler. The delimiter is akin to the boundary between a user process and an OS kernel. In particular, if the expression M is actually a value V, then the transaction is finished; if M has the form $C[\mathsf{suspend}\, V]$, then the transaction is suspended. These are the two cases of transaction expressions that are values.

In our core language, the only way to suspend a transaction is to evaluate the expression $\mathsf{suspend}\, V$ inside it. The value V here can be observed outside the transaction. We can think of $\mathsf{suspend}\, V$ as making an explicit system call with the argument V. In contrast, for transactions to provide secure isolation in JavaScript, every action with a side effect not recorded in the read/write set must implicitly suspend the current transaction, if any. This goal can be achieved by changing the JavaScript implementation (*e.g.,* bytecode interpreter) rather than any JavaScript code. The actions that should implicitly suspend include I/O operations and calls to DOM methods such as `document.write`. It is then up to the code outside the transaction to filter the actions defensively.

A metacontext D is a sequence of pairs of read/write sets RW and contexts C, which are

[2]The read set of the transaction does not include $\ell_2 \mapsto 25$ because reading data previously written by the same transaction, being of no concern outside the transaction, is not recorded in the read set.

the heap states and control states of a sequence of nested ongoing transactions. A metacontext is also an expression in which a subexpression next to be evaluated is replaced by a hole $\Box$.

We define two functions to help manipulate read/write sets. The partial function Read maps a metacontext and a location to a value, by looking up the location in the metacontext's read/write sets:

$$\mathrm{Read}(D[RW\{C\}], \ell) = \begin{cases} W(\ell) & \text{if } W(\ell) \text{ is defined,} \\ \mathrm{Read}(D, \ell) & \text{otherwise.} \end{cases}$$

The function Write combines two write sets W and W' into one, preferring entries in W' over those in W:

$$\mathrm{Write}(W, W')(\ell) = \begin{cases} W'(\ell) & \text{if } W'(\ell) \text{ is defined,} \\ W(\ell) & \text{otherwise.} \end{cases}$$

Finally, in Figure 5.4, we define a (small-step) transition relation $\rightsquigarrow$ between machine states. A machine state is just a transaction expression $RW\{M\}$; thus, we treat the entire machine as executing a top-level transaction (whose read set does not matter). In the transitions, we write $(x \mapsto V)M$ to denote the (capture-avoiding) substitution of V for x in M. The transition relation so defined is patently deterministic modulo the renaming of locations and variables. We denote the transitive closure of $\rightsquigarrow$ by $\rightsquigarrow^+$.

The introspect facility defined here is very simple: it only lets a policy observe whether a transaction is finished or suspended (correponding to `isSuspended` in Figure 5.2), and with what value (`getCause`, `getObject`, and `getArgs` in Figures 5.2). In our proposed implementation, transaction objects provide more information about their read/write sets, so a policy can check if they contain a given location (using `getWriteSet` and `checkMembership` in Figure 5.2) or enumerate their contents. This information can be used, for example, to see if the transaction has read any sensitive information, *e.g.,* cookies, that should not be leaked, or made any changes, *e.g.,* to global variables, that should not be committed.

More broadly speaking, the model of locations and variables in our lambda calculus is

$$
\begin{array}{ll}
D[n_1 + n_2] & \rightsquigarrow D[n] \\
\quad \text{where } n \text{ is the sum of } n_1 \text{ and } n_2 & \\
D[(\lambda x.\, M)V] & \rightsquigarrow D[(x \mapsto V)M] \\
D[RW\{C[\mathsf{commit}\, R'W'\{M\}]\}] & \rightsquigarrow D[RW''\{C[0]\}] \\
\quad \text{where } W'' = \mathrm{Write}(W, W') & \\
D[\mathsf{introspect}\,(RW\{M\})\,(x_1.M_1)\,(x_2.M_2)] & \rightsquigarrow D[(x_i \mapsto V)M_i] \\
\quad \text{where } M = V \text{ and } i = 1 \text{ or } M = C[\mathsf{suspend}\, V] \text{ and } i = 2 & \\
D[RW\{C[\mathsf{new}\, V]\}] & \rightsquigarrow D[RW'\{C[\ell]\}] \\
\quad \text{where } \ell \text{ is fresh and } W' = \mathrm{Write}(W, \{\ell \mapsto V\}) & \\
D[RW\{C[\mathsf{read}\, \ell]\}] & \rightsquigarrow D[R'W\{C[V]\}] \\
\quad \text{where } V = W(\ell) \text{ and } R' = R \text{ if } W(\ell) \text{ is defined,} & \\
\qquad V = \mathrm{Read}(D, \ell) \text{ and } R' = R \cup \{\ell \mapsto V\} \text{ otherwise} & \\
D[RW\{C[\mathsf{write}\, \ell\, V]\}] & \rightsquigarrow D[RW'\{C[V]\}] \\
\quad \text{where } W' = \mathrm{Write}(W, \{\ell \mapsto V\}) & \\
D[\mathsf{resume}\,(RW\{C[\mathsf{suspend}\, V]\})\, V'] & \rightsquigarrow D[RW\{C[V']\}]
\end{array}
$$

Figure 5.4: **The transition relation $\rightsquigarrow$ between states during a transaction evaluation.**

much simpler than JavaScript's, which involves, for example, looking up variables along scope chains and properties along prototype chains. These complications can be modeled without any fundamental difficulty—either using the Read and Write functions defined above, or by writing a JavaScript interpreter in our lambda calculus.

5.5.2 Examples

To illustrate the transition relation, we present some small example programs. For clarity, we write $\mathsf{var}\, x = M_1;\ M_2$ to abbreviate the expression $(\lambda x.\, M_2)M_1$. To express loops (which typical policies are), we also write $\mathsf{function}\, f(x)\, M$ to abbreviate the value

$$\lambda x.\, (\lambda y.\ \mathsf{var}\, f = \lambda x.\, yyx;\ \lambda x.\, M)(\lambda y.\ \mathsf{var}\, f = \lambda x.\, yyx;\ \lambda x.\, M)x.$$

The latter abbreviation has the crucial fixpoint property, that is,

$$
\begin{array}{c}
D[(\mathsf{function}\, f(x)\, M)V] \\
\rightsquigarrow^{+} D[(f \mapsto \mathsf{function}\, f(x)\, M)(x \mapsto V)M].
\end{array}
$$

Take for example the policy P_1, defined as the value

$$\mathsf{function}\, p(t)\ \mathsf{introspect}\, t\, (r.\, r)\, (a.\, p(\mathsf{resume}\, t\, (a+1))).$$

Ignoring the use of resume for the moment, suppose we apply this policy to the trivial transaction $\{\}\{\}\{3+4\}$ (that is, the expression $3+4$ delimited by an empty read set and an empty write set). This transaction immediately finishes with the result 7, which is observed by the policy due to $(r.\, r)$:

$$\begin{aligned}
& \{\}\{\}\{P_1(\{\}\{\}\{3+4\})\} \\
\rightsquigarrow\ & \{\}\{\}\{P_1(\{\}\{\}\{7\})\} \\
\rightsquigarrow^+\ & \{\}\{\}\{\mathsf{introspect}\, (\{\}\{\}\{7\}) \\
& \qquad\qquad\qquad\qquad (r.\, r) \\
& \qquad\qquad\qquad\qquad (a.\, P_1(\mathsf{resume}\, t\, (a+1)))\} \\
\rightsquigarrow\ & \{\}\{\}\{7\}
\end{aligned}$$

Suppose that ℓ is a location shared between the host application and the contained transaction. Even if the transaction reads and writes ℓ in the course of its computation, as long as the policy does not commit the transaction—which P_1 does not—the changes will not be reflected in the global write set. For example, the transaction below increments the content of ℓ and returns the result:

$$\begin{aligned}
& \{\}\{\}\{\mathsf{var}\, x = \mathsf{new}\, 1;\ P_1(\{\}\{\}\{\mathsf{write}\, x\, (\mathsf{read}\, x+1)\})\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{\mathsf{var}\, x = \ell;\ P_1(\{\}\{\}\{\mathsf{write}\, x\, (\mathsf{read}\, x+1)\})\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{P_1(\{\}\{\}\{\mathsf{write}\, \ell\, (\mathsf{read}\, \ell+1)\})\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{P_1(\{\ell \mapsto 1\}\{\}\{\mathsf{write}\, \ell\, (1+1)\})\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{P_1(\{\ell \mapsto 1\}\{\}\{\mathsf{write}\, \ell\, 2\})\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{P_1(\{\ell \mapsto 1\}\{\ell \mapsto 2\}\{2\})\} \\
\rightsquigarrow^+\ & \{\}\{\ell \mapsto 1\}\{\mathsf{introspect}\, (\{\ell \mapsto 1\}\{\ell \mapsto 2\}\{2\}) \\
& \qquad\qquad\qquad\qquad (r.\, r) \\
& \qquad\qquad\qquad\qquad (a.\, P_1(\mathsf{resume}\, t\, (a+1)))\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{2\}
\end{aligned}$$

The finished transaction has the read set $\{\ell \mapsto 1\}$ and the write set $\{\ell \mapsto 2\}$. They are discarded by introspect in the policy, even though the result 2 of the transaction, computed using them, is retained.

The same read/write sets track even locations created and used solely within the transaction. For example, we can move the variable x into the transaction above and obtain the same result 2:

$$\begin{aligned}
& \{\}\{\}\{P_1(\{\}\{\}\{\mathsf{var}\, x = \mathsf{new}\, 1;\ \mathsf{write}\, x\, (\mathsf{read}\, x + 1)\})\} \\
\rightsquigarrow\ & \{\}\{\}\{P_1(\{\}\{\ell \mapsto 1\}\{\mathsf{var}\, x = \ell;\ \mathsf{write}\, x\, (\mathsf{read}\, x + 1)\})\} \\
\rightsquigarrow^+\ & \{\}\{\}\{P_1(\{\}\{\ell \mapsto 2\}\{2\})\} \\
\rightsquigarrow^+\ & \{\}\{\}\{2\}
\end{aligned}$$

Although we do not model garbage collection here (so the write-set entry $\ell \mapsto 2$ above persists until the transaction finishes), read/write sets should be subject to garbage collection. In other words, they should refer to locations only *weakly*.

To allow the transaction's write set to take global effect, the policy must commit the transaction explicitly, as in the following policy P_2:

$$\begin{aligned}
&\mathsf{function}\, p(t)\ \mathsf{introspect}\, t \\
&\qquad\qquad (r.\ \mathsf{var}\, z = \mathsf{commit}\, t;\ r) \\
&\qquad\qquad (a.\, p(\mathsf{resume}\, t\, (a + 1)))
\end{aligned}$$

(The variable z above is just to receive the dummy result 0 returned by commit.) Applying P_2 to the same transaction modifies ℓ globally to 2:

$$\begin{aligned}
& \{\}\{\}\{\mathsf{var}\, x = \mathsf{new}\, 1;\ P_2(\{\}\{\}\{\mathsf{write}\, x\, (\mathsf{read}\, x + 1)\})\} \\
\rightsquigarrow^+\ & \{\}\{\ell \mapsto 1\}\{\mathsf{introspect}\, (\{\ell \mapsto 1\}\{\ell \mapsto 2\}\{2\}) \\
& \qquad\qquad (r.\, \mathsf{var}\, z = \mathsf{commit} \\
& \qquad\qquad\qquad\qquad (\{\ell \mapsto 1\}\{\ell \mapsto 2\}\{2\}); \\
& \qquad\qquad r) \\
& \qquad\qquad (a.\, P_2(\mathsf{resume}\, t\, (a + 1)))\}
\end{aligned}$$

$$\rightsquigarrow \{\}\{\ell \mapsto 1\}\{\mathsf{var}\, z = \mathsf{commit}(\{\ell \mapsto 1\}\{\ell \mapsto 2\}\{2\});\, 2\}$$
$$\rightsquigarrow \{\}\{\ell \mapsto 2\}\{\mathsf{var}\, z = 0;\, 2\}$$
$$\rightsquigarrow \{\}\{\ell \mapsto 2\}\{2\}$$

Hence, we have no rollback operation—to roll back a transaction is simply to never commit it.

Finally, we illustrate the use of suspend and resume using the transaction

$$T = \mathsf{var}\, z = \mathsf{write}\, \ell\, (\mathsf{suspend}\, (\mathsf{read}\, \ell));$$
$$\mathsf{write}\, \ell\, (\mathsf{suspend}\, (\mathsf{read}\, \ell)).$$

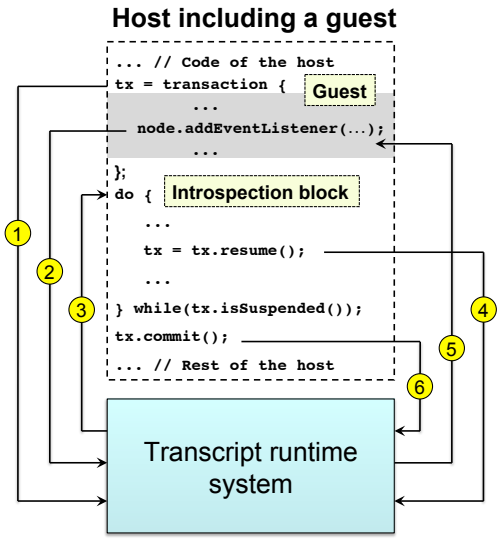
Twice in a row, this transaction sends the content of ℓ to the host as a request and puts the host's response back into ℓ. The policies P_1 and P_2 above implement an integer incrementation service, so applying P_1 or P_2 to T increments the content of ℓ twice in a row:

$$\begin{aligned}
& \{\}\{\ell \mapsto 1\}\{P_1(\{\}\{\}\{T\})\} \\
\rightsquigarrow^+\ & \{\}\{\ell \mapsto 1\}\{\mathsf{introspect}\, T'\, (r.\, r)\, (a.\, P_1(\mathsf{resume}\, T'\, (a{+}1)))\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{P_1(\mathsf{resume}\, T'\, (1+1))\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{P_1(\mathsf{resume}\, T'\, 2)\} \\
\rightsquigarrow\ & \{\}\{\ell \mapsto 1\}\{P_1(\{\ell \mapsto 1\}\{\}\{\mathsf{var}\, z = \mathsf{write}\, \ell\, 2; \\
& \qquad\qquad\qquad\qquad \mathsf{write}\, \ell\, (\mathsf{suspend}\, (\mathsf{read}\, \ell))\})\} \\
\rightsquigarrow^+\ & \{\}\{\ell \mapsto 1\}\{P_1(\{\ell \mapsto 1\}\{\ell \mapsto 2\}\{\mathsf{write}\, \ell\, 3\})\} \\
\rightsquigarrow^+\ & \{\}\{\ell \mapsto 1\}\{3\}
\end{aligned}$$

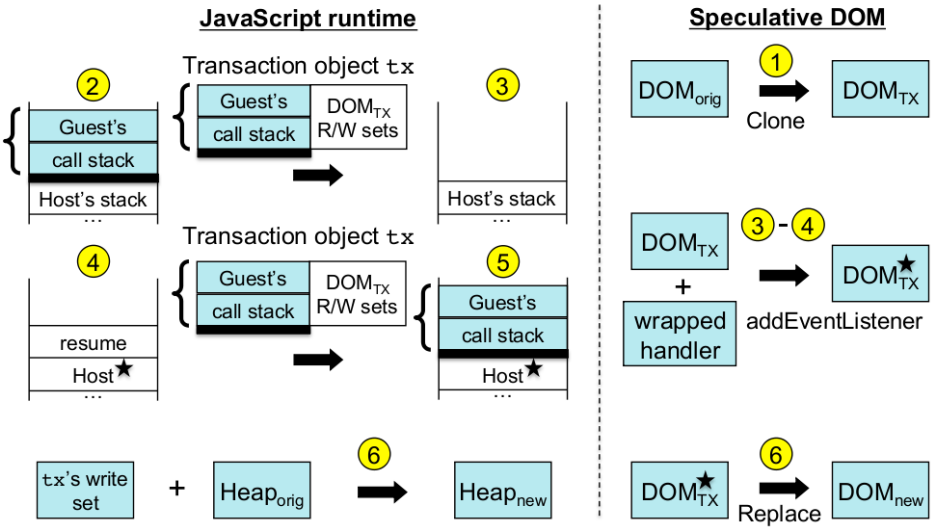
where T' is short for $\{\ell \mapsto 1\}\{\}\{\mathsf{var}\, z = \mathsf{write}\, \ell\, (\mathsf{suspend}\, 1);$
$\mathsf{write}\, \ell\, (\mathsf{suspend}\, (\mathsf{read}\, \ell))\}$.

5.6 Design of Transcript

We now describe the design of Transcript's mechanisms using Figure 5.5, which summarizes the workflow of a Transcript-enhanced host. The figure shows the operation of the Transcript runtime system at key points during the execution of the host, which has included an untrusted



(a) Locations of traps/returns to/from Transcript.



(b) Corresponding actions within the Transcript runtime system for a trap/return.

Figure 5.5: **Workflow of a Transcript-enhanced host.** Part (a) of the figure shows a host enclosing a guest within a transaction and an inlined introspection block, while part (b) shows the JavaScript runtime and the DOM subsystem. The labels ①-⑥ in the figure show: ① the host's DOM being cloned at the start of the transaction, ② the host's call stack before a call that suspends the transaction, ③ the call stack after suspension, ④ the host's call stack when the transaction is about to resume; the speculative DOM has been updated with the requested changes, ⑤ the host's call stack just after resumption, ⑥ the commit of the transaction, which copies all speculative changes to the host's DOM and JavaScript heap. The thick lines on the call stacks denote transaction delimiters. Arrows show control transfer from the transaction to the iblock and back.

guest akin to the one in Figure 5.1 using a transaction.

When a transaction begins execution, Transcript first provides the transaction with its private copy of the host's DOM tree. It does so by cloning the current state of the host's DOM, including any event handlers associated with the nodes of the DOM (① in Figure 5.5). When a guest references nodes in the host's DOM, Transcript redirects these references to the corresponding nodes in the transaction's private copy of the DOM.

Next, the Transcript runtime pushes a *transaction delimiter* on the JavaScript call stack. Transcript places the activation records of methods invoked within the transaction above this delimiter. It also records the locations of JavaScript objects accessed/modified within the transaction in read/write sets. If the transaction executes an external operation, the runtime suspends the transaction. To do so, it creates a transaction object and (a) initializes the object with the transaction's read/write sets; (b) pops all the activation records on the JavaScript call stack until the topmost transaction delimiter; (c) stores these activation records in the transaction object; (d) saves the program counter; and (e) sets the program counter to immediately after the end of the transaction, *i.e.,* the start of the iblock (steps ② and ③ in Figure 5.5).

The iblock logically extends from the end of the transaction to the last `resume` or `commit` call on the transaction object (*e.g.,* lines **A**–**T** in Figure 5.2). The iblock can access the transaction object and its read/write sets to make policy decisions. If the iblock invokes `resume` on a suspended transaction, the Transcript runtime (a) pushes a transaction delimiter on the current JavaScript call stack; (b) pushes the activation records saved in the transaction object; and (c) restores the program counter to its saved value. Execution therefore resumes from the statement following the external operation (see ④ and ⑤). If the iblock invokes `commit` instead, the Transcript runtime updates the JavaScript heap using the values in the transaction object's write set. The `commit` operation also replaces the host's DOM with the cloned DOM (step ⑥).

The Transcript runtime behaves in the same way even when transactions are nested: Transcript pushes a new delimiter on the JavaScript call stack for each level of nesting encountered at runtime. Each suspend operation only pops activation records until the topmost delimiter on the stack. Nesting is important when a guest itself wishes to confine code that it does not

trust. This situation arises when a host includes a guest from a first-tier advertising agency (`1sttier.com`), which itself includes code from a second-tier agency (`2ndtier.com`). Whether the host confines the advertisement using an *outer* transaction, `1sttier.com` may itself confine code from `2ndtier.com` using an *inner* transaction using its own security policies. If code from `2ndtier.com` attempts to modify the DOM, that call suspends and traps to the iblock defined by `1sttier.com`. If this iblock attempts to modify the DOM on behalf of `2ndtier.com`, the outer transaction suspends in turn and passes control to the host's iblock. In effect, the DOM modification succeeds only if it is permitted at *each* level of nesting.

5.6.1 Components of an Iblock

As discussed in Section 5.4, an iblock consists of two parts: a *host-specific* part, which codifies the host's policies to mediate guests, and a *mandatory* part, which contains functionality that is generic to all hosts. In our implementation, we have encoded the second part as a JavaScript library (`libTranscript`) that can simply be included into the iblock of a host. This mandatory part implements two functionalities: *gluing execution contexts* and *generating wrappers for event handlers*.

Gluing Execution Contexts

Guests often use `document.write` or similar calls to modify the host's DOM, as shown on line **9** of Figure 5.1. When such guests execute within a transaction, the `document.write` call traps to the iblock, which must complete the call on behalf of the guest and render the HTML in the cloned DOM. However, the HTML code in `document.write` may contain scripts, *e.g.,* `document.write('<script src = code.js>')`. The execution of `code.js`, having been triggered by the guest, must then be mediated by the same security policy that governs the guest.

Thus, `code.js` should be executed in the same context as the transaction where the guest executes. To achieve this goal, the mandatory part of the iblock encapsulates the content of `code.js` into a function and uses a builtin `glueresume` method of the transaction object

to instruct the Transcript runtime to invoke this function when it resumes the suspended transaction. The net effect is similar to fetching and inlining the content of `code.js` into the transaction. We call this operation *gluing*, because it glues the code in `code.js` to that of the guest.

To implement gluing, the iblock must recognize that the `document.write` includes additional scripts. This in turn requires the iblock to parse the HTML argument to `document.write`. We therefore exposed the browser's HTML parser through a new `document.parse` API to allow HTML (and CSS) parsing in iblocks. This API accepts a HTML string argument, such as the argument to `document.write`, and parses it to recognize `<script>` elements and other HTML content. It also recognizes inline event-handler registrations, so that they can be wrapped as described in Section 5.6.1. When the iblock invokes `document.parse` (in Figure 5.2, it is invoked within the call to `WriteToTxDOM` on line **M**), the parser creates new functions that contain code in `<script>` elements. It returns these functions to the host's iblock, which can then invoke them by gluing. The parser also renders other (non-script) HTML content in the cloned DOM.

Guest operations involving `innerHTML` are handled similarly. Transcript suspends a guest that attempts an `innerHTML` operation, parses the new HTML code for any scripts, and glues their execution into the guest's context.

Generating Wrappers for Event Handlers

Guests executing within a transaction may attempt to register functions to handle asynchronous events. For example, line **8** in Figure 5.1 registers `displayAds` as an `onMouseOver` handler. Because `displayAds` is guest code, it is important to associate it with the iblock for the transaction that registered it and to subject it to the same policy checks. Transcript does so by creating a new function `tx_displayAds` that *wraps* `displayAds` within a transaction guarded by the same iblock, and registering `tx_displayAds` as the event handler for the `onMouseOver` event.

To this end, the mandatory part of the iblock includes creating wrappers (such as `tx_displayAds`) for event handlers. When the guest executes a statement such as

```
1  tx_clkhandler = function(evt) {
2    evttx = transaction { node.evtH (evt); }
3    iblock_func (evttx);
4  }
```

Figure 5.6: **Example code snippet to generate transactional event handlers.**

`elem.addEventListener(...)`, it would trap to the iblock, which can then examine the arguments to this call and create a wrapper for the event handler. Guests can alternatively use `document.write` calls to register event handlers *e.g.,* `document.write ('<div onMouseOver="displayAds();">')`. In this case, the iblock recognizes that an event handler is being registered by parsing the HTML argument of the `document.write` call (using the `document.parse` API) when it suspends, and wraps the call. Firefox currently supports three event-handling models [172]. For each model, the goal of the wrapper generator is to obtain a reference to the handler being registered, and wrap it suitably. We describe the three models briefly and discuss how Transcript obtains a reference to the handler in each case.

(1) The *DOM-level 0/Traditional* model registers an event handler for a DOM node, *e.g.,* a `<div>` element, as follows: `node.onclick = clkhandler`. Here `node` represents the `<div>` element and `clkhandler` is registered as a handler for `onclick` events. Transcript modifies the interpreter to suspend transactions that change properties containing event handlers, such as `onclick` and `onload`. Once the transaction suspends, the iblock obtains a reference to `clkhandler`.

(2) The *DOM-level 0/Inline* model registers an event handler using code such as: `document.write("<div onclick="/*handler code*/>")`, which sets an attribute (*e.g.,* `onclick`) of the DOM node. Transcript handles such cases by suspending the execution of `document.write`. The argument to this call is HTML code, which the transaction's iblock parses to obtain a reference to the event handler.

(3) The *DOM-level 2* model registers an event handler as follows: `node.addEventListener("click", clkhandler, false)`. Transcript suspends `addEventListener`, thereby allowing the iblock to obtain a reference to `clkhandler`.

```
1  var tx = transaction {
2    ...  //code that suspends ...
3    for (var x in this) {
4      if (this[x] instanceof Tx_obj) txref = this[x];
5    }; txref.getWriteSet = function() { };
6  }
```

Figure 5.7: **Example of a third party JavaScript code that implements a reference leak.** The `tx` object is created and attached to `this` when the code suspends on line **2**.

In addition to obtaining a reference to the event handler, the iblock also obtains a reference to `node`, which is the DOM node for which the handler was being registered. The iblock then initializes a new property `node.evtH` with `clkhandler`, and defines a new function `tx_clkhandler` as shown in Figure 5.6, which it registers as the event handler. Here, `iblock_func` is a function that contains the iblock itself, while `evt` is a JavaScript object that the browser uses to denote the event. As a result of this transformation, `tx_clkhandler` is invoked when the `onclick` event is triggered, which then executes `clkhandler` within a transaction, thereby allowing Transcript to mediate its operation as well.

Besides event handlers, JavaScript supports other constructs for asynchronous execution: AJAX callbacks, which execute upon receiving network events (`XMLHttpRequest`), and features, such as `setTimeOut` and `setInterval`, that trigger code execution based upon timer events. The mandatory part of the iblock also handles these constructs by wrapping callbacks as just described.

5.6.2 Hiding Sensitive Variables

The iblock of a transaction checks the guest's actions against the host's policies. These policies are themselves encoded in JavaScript, and may use methods and variables (*e.g.,* `tx`, `tocommit` and `builtins` in Figure 5.1) that must be protected from the guest. Without precautions, the guest can use JavaScript's extensive reflection capabilities to tamper with these sensitive variables. Figure 5.7 presents an example of one such attack, a reference leak, where the malicious guest obtains a reference to the `tx` object by enumerating the properties of the `this` object, and redefines the method `tx.getWriteSet` speculatively. As presented, example in Figure 5.1 is vulnerable to such a reference leak.

To protect such sensitive variables, we adopt a defense called *variable hiding* that eliminates the possibility of leaks by construction. This technique mandates that guests be placed outside the scope of the iblock's variables, such as `tx`. The basic idea is to place the guest and the iblock in separate, lexically scoped functions, so that variables such as `tx`, `tocommit` and `builtins` are not accessible to the guest. By so hiding sensitive variables from the guest, this defense prevents reference leaks. Figure 5.10 illustrates this defense after introducing some more details of our implementation.

5.7 Security Assurances

Transcript's ability to protect hosts from untrusted guests depends on two factors: (a) the assurance that a guest cannot subvert Transcript's mechanisms, *i.e.,* the robustness of the trusted computing base; and (b) host-specific policies used to mediate guests.

5.7.1 Trusted Computing Base

Transcript's trusted computing base (TCB) consists of the runtime component implemented in the browser and the mandatory part of the host's iblock. The TCB provides the following security properties: (a) *complete mediation*, *i.e.,* control over all JavaScript and external operations performed by a guest; and (b) *isolation*, *i.e.,* the ability to confine the effects of the guest.

- **Complete mediation.** The Transcript runtime and the mandatory part of the host's iblock together ensure complete mediation of guest execution. The runtime: (a) records all guest accesses to the host's JavaScript heap in the corresponding transaction's read/write sets; (b) causes a trap to the host's iblock when the guest attempts an external operation; and (c) redirects all guest references to the host's DOM to the cloned DOM. The mandatory part of the iblock, consisting of wrapper generators and the HTML parser, ensures that any additional code fetched by the guest or scheduled for later execution (*e.g.,* event handlers or callbacks for `XMLHttpRequest`) will itself be enclosed within transactions mediated by the same iblock. This process recurs so that the host's policies mediate all guest code, even event handlers installed by callbacks of event handlers.

- **Isolation.** Transcript isolates guest operations using speculative execution. It records changes to the host's JavaScript heap within the guest transaction's write set, and changes to the host's DOM within the cloned DOM. The host then has the opportunity to review these speculative changes within its iblock and ensure that they conform to its security policies. Observe that a suspended/completed transaction may provide the host with references to objects modified by the guest, *e.g.,* in Figure 5.1, a reference to `elem` is passed to the iblock via the `getObject` API. Speculative execution ensures that if the transaction has not yet been committed, then accesses to the object's methods and fields via this reference will still resolve to their values at the beginning of the transaction. Thus, for instance, a call to the `toString` method of the `elem` object in the iblock of Figure 5.1 would still work as intended if even if the guest had redefined this method within the transaction. Note that variables hidden from the guest cannot even be *speculatively* modified, thereby automatically isolating them from the guest.

Together, the above properties ensure the following invariant: At the point when a transaction suspends or completes execution and is awaiting inspection by the host's iblock, none of the host's JavaScript objects or its DOM would have been modified by the guest. Further, host variables hidden from the guest will not be modified even after the transaction has committed. Overall, executing a transaction never incurs any side effect, and any side effect that would be incurred by committing a transaction can be first vetted by inspecting the transaction.

5.7.2 Whitelisting for Host Policies

Hosts can import the speculative changes made by a guest after inspecting them against their security policies. Even though complete mediation and isolated execution ensure that the core *mechanisms* of Transcript cannot be subverted by guest execution (*i.e.,* they ensure that all of the guest's speculative actions will be available for inspection by the host), ability of the host to isolate itself from the guest ultimately depends on its *policies*.

Host policies are necessarily domain-specific and have to be written manually in our current prototype. Though our experiments (Section 5.9.4) suggest that the effort required to write policies in Transcript is comparable to that required in other systems, writing policies

is admittedly a difficult exercise and further research is needed to develop tools for policy authors to debug/verify the completeness of their policies. We suggest that iblock authors should employ a whitelist that specifies the host objects that can legitimately be modified by the guest and reject attempts to modify objects outside the whitelist. This guideline may cause false positives if the whitelist is not comprehensive. For example, both `window.location` and `window.location.href` can be used to change the location field of the host, but a whitelist that includes only one will reject guests that modify guest location using the other. Nevertheless, whitelisting allows hosts to be conservative when allowing guests to modify their objects.

5.8 Implementation in Firefox

We implemented Transcript by modifying Firefox (version 3.7a4pre). Overall, our prototype adds or modifies about 6,400 lines of code in the browser. The bulk of this section describes Transcript's enhancements to SpiderMonkey (Firefox's JavaScript interpreter) (Section 5.8.1) and its support for speculative DOM updates (Section 5.8.2). We also discuss Transcript's support for conflict detection (Section 5.8.3) and the need to modify the `<script>` tag (Section 5.8.4).

5.8.1 Enhancements to SpiderMonkey

Our prototype enhances SpiderMonkey in five ways:

- *Transaction objects.* We added a new class of JavaScript objects to denote transactions. This object stores pointers to the read/write sets, activation records of the transaction, and to the cloned DOM. It implements the builtin methods shown in Figure 5.1.

- *A `transaction` keyword.* We added a `transaction` keyword to the syntax of JavaScript. When the Transcript-enhanced JavaScript parser encounters this keyword, it (a) compiles the body of the transaction into an anonymous function; (b) inserts a new instruction, JSOP_BEGIN_TX, into the generated bytecode to signify the start of a transaction; and (c) inserts code to invoke the anonymous function. The transaction ends

API	Description
`getReadSet`	Exports transaction's read set to JavaScript.
`getWriteSet`	Exports transaction's write set to JavaScript.
`getTxDocument`	Returns a reference to the speculative `document` object.
`isSuspended`	Returns `true` if the transaction is suspended.
`getCause`	Returns cause of a transaction suspend.
`getObject`	Returns object reference on which a suspension was invoked.
`getArgs`	Returns set of arguments involved in a transaction suspend.
`resume`	Resumes suspended transaction.
`glueresume`	Resumes suspended transaction and glues execution contexts.
`isDOMConflict`	Checks for conflicts between the host's and cloned DOM.
`isHeapConflict`	Checks for conflicts between the host and guest heaps.
`commit`	Commits changes to host's JavaScript heap and DOM.

Table 5.1: **Key APIs defined on the transaction object.**

when the anonymous function completes execution. Finally, the anonymous function returns a transaction object when it suspends or completes execution.

- *Read/write sets.* Transcript adds read/write set-manipulation to the interpretation of several JavaScript bytecode instructions. We enhanced the interpreter so that each bytecode instruction that accesses or modifies JavaScript objects additionally checks whether its execution is within a transaction (*i.e.,* if an unfinished JSOP_BEGIN_TX was previously encountered in the bytecode stream). If so, the execution of the instruction also logs an identifier denoting the JavaScript object (or property) accessed/modified in its read/write sets, which we implemented using hash tables. We used SpiderMonkey's identifiers for JavaScript objects; references using aliases to the same object will return the same identifier.

- *Suspend.* We modified the interpreter's implementation of bytecode instructions that perform external operations and register event handlers to suspend when executed within a transaction. The suspend operation and the builtin `resume` function of transaction objects are implemented as shown in Figure 5.5. We also introduced a `suspend` construct that allows hosts to customize transaction suspension. Hosts can include this construct within a transaction (before including guest code) to register custom suspension points. The call `suspend [obj.foo]` suspends the transaction when it invokes `foo` (if it is a method) or attempts to read from or write to the property `foo` of `obj`.

- *Garbage Collection.* We interfaced Transcript with the garbage collector to traverse and mark all heap objects that are reachable from live transaction objects. This avoids any inadvertent garbage collection of objects still reachable from suspended transactions that could be resumed in the future.

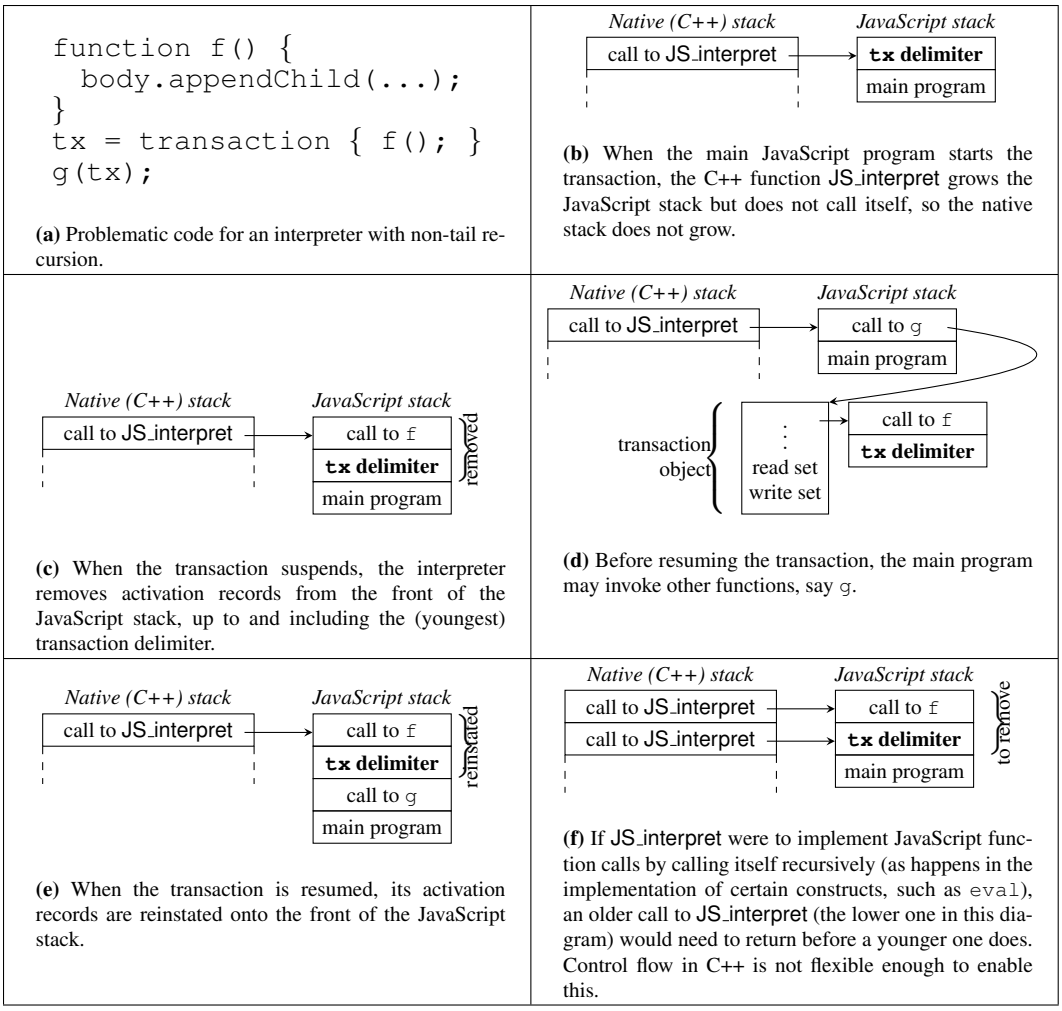
Integrating these changes into a legacy JavaScript engine proved to be a challenging exercise. We now describe how our implementation addressed one such challenge, non-tail recursive calls in SpiderMonkey.

Non-tail-recursive Interpreters

A key challenge in enhancing a legacy JavaScript interpreter, such as SpiderMonkey, with support for transactions is in how the interpreter uses recursion. To support the suspend/resume mechanism for switching control flow between a transaction and its iblock, the interpreter must not accumulate any activation records in its native stack (*e.g.,* the C++ stack, for SpiderMonkey) between when a transaction starts and when it suspends. In particular, the interpreter must not represent JavaScript function calls by C++ function calls. The same issue also arises when a compiler or JIT interpreter is used to turn JavaScript code into machine code.

To illustrate this point, consider SpiderMonkey, which implements the bytecode interpreter in C++. The main entry point to the bytecode interpreter is the C++ function JS_interpret, which maintains the JavaScript stack as a linked list of activation records, each of which is a C++ structure. When one function calls another in JavaScript, the JS_interpret function does not call itself in C++; instead, it adds a new activation record to the front of the linked list and continues with the same bytecode interpreter loop as before. Similarly, when a function returns to another in JavaScript, JS_interpret does not return in C++; instead, it removes an old activation record from the front of the linked list and continues with the same bytecode interpreter loop as before. For the most part, SpiderMonkey does not represent JavaScript calls by C++ calls.

The fact that SpiderMonkey does not represent JavaScript calls by native calls helps us add transactions to it without making invasive changes, as the following example illustrates. Suppose a transaction invokes a function `f` that suspends for some reason, *e.g.,* in Figure 5.8(a), the



(a) Problematic code for an interpreter with non-tail recursion.

(b) When the main JavaScript program starts the transaction, the C++ function JS_interpret grows the JavaScript stack but does not call itself, so the native stack does not grow.

(c) When the transaction suspends, the interpreter removes activation records from the front of the JavaScript stack, up to and including the (youngest) transaction delimiter.

(d) Before resuming the transaction, the main program may invoke other functions, say `g`.

(e) When the transaction is resumed, its activation records are reinstated onto the front of the JavaScript stack.

(f) If JS_interpret were to implement JavaScript function calls by calling itself recursively (as happens in the implementation of certain constructs, such as `eval`), an older call to JS_interpret (the lower one in this diagram) would need to return before a younger one does. Control flow in C++ is not flexible enough to enable this.

Figure 5.8: **Native versus JavaScript call stacks during transaction suspend/resume.** The implementation of suspend/resume assumes an interpreter without non-tail recursion.

function `f` calls `appendChild`. If the C++ call to JS_interpret that executes the transaction were not same as the one that executes the called function `f`, then the former, although older, would have to return before the latter returns. As detailed in 5.8, the former has to return when suspending the transaction, whereas the latter has to return when resuming the transaction. Even exception handling in C++ does not allow such control flow.

Unfortunately, JS_interpret in SpiderMonkey does call itself in a few situations. For example, it handles JavaScript's `eval` in this way, and the problem of the C++ stack in Figure 5.8(f) does arise if we replace the `body.appendChild(...)` of Figure 5.8(a) by `eval("body.appendChild(...)")`. One way to solve this problem requires applying

the continuation-passing-style transformation to the interpreter to put it into tail form, *i.e.,* convert all recursive calls to JS_interpret to tail calls. However, this transformation is invasive, especially if done manually on legacy interpreters.

Transcript uses a less invasive mechanism to enable suspend/resume in SpiderMonkey. This mechanism is similar in functionality to gluing (see Section 5.6.1), and we explain it with an example. Consider the `eval` construct, whose functionality is to parse its input string, compile it into bytecode, and then execute the bytecode as usual. Because only the last step, *i.e.,* that of executing the bytecode, can suspend, we simply changed the behavior of `eval` so that, if invoked inside a transaction, it suspends the transaction right away. The iblock of the transaction can then compile the string into bytecode and include the bytecode into the execution of the transaction. This is achieved by adding a new activation record to the front of the transaction's JavaScript stack and modifying the program counter to execute this code when the transaction resumes. When the suspended transaction resumes, it transfers control to the `eval`ed code, which can freely suspend. Besides `eval`, our current Transcript prototype also implements gluing for `document.write` (as discussed in Section 5.6.1) and JavaScript builtins `call` and `apply`, which make non-tail recursive calls to JS_interpret.

5.8.2 Supporting Speculative DOM Updates

Transcript provides each executing transaction with its private copy of the host's document structure and uses this copy to record all DOM changes made by guest code. This section presents notable details of the implementation of Transcript's DOM subsystem.

Transcript constructs a replica of the host's DOM when it encounters a JSOP_BEGIN_TX instruction in the bytecode stream. It clones nodes in the host's DOM tree, and iterates over each node in the host's DOM to copy references to any event handlers and dynamically-attached JavaScript properties associated with the node. If a guest attempts to modify an event handler associated with a node, the reference is rewritten to point to the function object in the transaction's write set.

Crom [114] also implemented DOM cloning for speculative execution (albeit not for the purpose of mediating untrusted code). Unlike Crom, which implemented DOM cloning as a

JavaScript library, Transcript implements cloning in the browser itself. This feature simplifies several issues that Crom's designers faced (*e.g.,* cloning DOM-level 2 event handlers) and also allows efficient cloning.

When a guest references a DOM node within a transaction, Transcript transparently redirects this reference to the cloned DOM. It achieves this goal by modifying the browser to tag each node in the host's DOM with a unique identifier (`uid`). During cloning, Transcript assigns each node in the cloned DOM the same `uid` as its counterpart in the host's DOM. When the guest attempts to access a DOM node, Transcript retrieves the `uid` of the node and walks the cloned DOM for a match. We defined a `getElementByUID` API on the `document` object to return a node with a given `uid`.

If the guest's operations conform to the host's policies, the host commits the transaction, upon which Transcript replaces the host's DOM with the transaction's copy of the DOM, thereby making the guest's speculative changes visible to the host.

5.8.3 Conflict Detection

When a host decides to commit a transaction, Transcript will replace the host's DOM with the guest's DOM. Objects on the host's heap are also overwritten using the write set of the guest's transaction. During replacement, care must be taken to ensure that the host's state is consistent with the guest's state. Consider, for instance, a guest that performs an `appendChild` operation on a DOM node (say node `N`). This operation causes a new node to be added to the cloned DOM, and also suspends the guest transaction. However, the host may delete node `N` before resuming the transaction; upon resumption, the guest continues to update a stale copy of the DOM (*i.e.,* the cloned version). When the transaction commits, the removed DOM node will be added to the host's DOM.

Transcript adds the `isDOMConflict` and `isHeapConflict` APIs to the transaction object, which allow host developers to register conflict detection policies. When invoked in the host's iblock, the `isDOMConflict` API invokes the conflict detection policy on each DOM node speculatively modified within the transaction (using the transaction's write set to identify nodes that were modified). The `isHeapConflict` API likewise checks that the state of the

```
1  function hasParent(txNode) {
2    var parent = txNode.parentNode;
3    if (document.getElementByUID(parent.uid) != null)
4      return true;
5    else return false;
6  } ...
7  // tx is the transaction object
8  var isAllowed = tx.isDOMConflict(hasParent);
```

Figure 5.9: **Example code snippet to handle conflict detection in Transcript.**

host's heap matches the state of the guest's heap at the start of the transaction. The snippet in Figure 5.9 shows one example of such a conflict detection policy (using `isDOMConflict`) encoded in the host's iblock that verifies that each node speculatively modified by the guest (`txNode`) has a parent in the host's DOM.

While Transcript provides the core *mechanisms* to detect transaction conflicts, it does not dictate any *policies* to resolve them. The host must resolve such conflicts within the application-specific part of its iblocks.

5.8.4 The `<script>` Tag

The examples presented thus far show hosts including guest code by inlining it within a transaction. However, hosts typically include guests using `<script>` tags, *e.g.,* `<script src="http://untrusted.com/guest.js">`. Transcript also supports code inclusion using `<script>` tags. To do so, it extends the `<script>` tag so that the fetched code can be encapsulated in a function rather than run immediately. The host application can use the modified `<script>` tag as: `<script src="http://untrusted.com/guest.js" func="foobar">`. This tag encapsulates the code in `foobar`, which the host can then invoke within a transaction.

By itself, this modification unfortunately affects the scope chain in which the fetched code is executed. JavaScript code included using a `<script>` tag expects to be executed in the global scope of the host, but the modified `<script>` tag would put the fetched code in the scope of the function specified in the `func` attribute (*e.g.,* `foobar`).

We addressed this problem using a key property of `eval`. The ECMAScript standard [28,

Section 10.4.2] specifies that an *indirect* `eval` (*i.e.,* via a reference to the `eval` function) is executed in the global scope. We therefore extracted the body of the compiled function `foobar` and executed it using an indirect `eval` call within a transaction (see Figure 5.10). This transformation allowed all variables and functions declared in the function `foobar` to be speculatively attached to the host's global scope.

5.9 Evaluation

We evaluated four aspects of Transcript. First, in Section 5.9.1 we studied the applicability of Transcript to real-world guests, which varied in size from about 1,400 to 7,500 lines of code. Second, we show in Section 5.9.2 that a host that uses Transcript can protect itself and recover gracefully from malicious and buggy guests. Third, we report a performance evaluation of Transcript in Section 5.9.3. Last, in Section 5.9.4, we study the complexity of writing policies for Transcript. All experiments were performed with Firefox v3.7a4pre on a 2.33Ghz Intel Core2 Duo machine with 3GB RAM and running Ubuntu 7.10.

5.9.1 Case Studies on Guest Benchmarks

To evaluate Transcript's applicability to real-world guests, we experimented with five JavaScript applications, shown in Table 5.2. For each guest benchmark in Table 5.2, we played the role of a host developer attempting to include the guest into the host, *i.e.,* we created a Web page and included the code of the guest into the page using `<script>` tags. Most of the guests were implemented in several files; the `<script>` column in Table 5.2 shows the number of `<script>` tags that we had to use to include the guest into the host. We briefly describe three of these guest benchmarks and the domain-specific policies that were implemented for each iblock.

(1) JavaScript Menu is a standalone widget that implements pull-down menus. Figure 5.10 shows how we confined JavaScript Menu using Transcript. The iblock for JavaScript menu enforced a policy that disallowed the guest from accessing the network (`XMLHttpRequest`) or domain cookies.

```
1  <script src="jsMenu.js" func="menu"></script>
2  <script src="libTranscript.js></script>
3  <script>(function () {
4    var to_commit = true, e = eval;     // indirect eval
5    var tx = transaction { e(getFunctionBody(menu)); }
6    to_commit = gotoIblock(tx);
7    if(to_commit) tx.commit();
8  })(); </script>
```

Figure 5.10: **Code snippet for confining JavaScript Menu benchmark using Transcript.** This figure illustrates several concepts: (a) lines **1** and **5** demonstrate the enhanced `<script>` tag and the host's use of indirect `eval` to include the guest, which is compiled into a function (called `menu`; line **1**) (Section 5.8.4). `getFunctionBody` extracts the code of the function `menu`; (b) line **3** implements variable hiding (Section 5.6.2), making `tx` invisible to the guest; (c) our supporting library `libTranscript` (line **2**) implements the mandatory part of the iblock and is invoked from `gotoIblock`.

	Benchmark	Size (LoC)	`<script>` tags
1	JavaScript Menu [14]	1,417	1
2	Picture Puzzle [131]	1,709	3
3	GoogieSpell [129]	2,671	4
4	GreyBox [130]	2,338	7
5	Color Picker [13]	7,543	6

Table 5.2: **List of macrobenchmarks isolated using Transcript.** We used transactions to isolate each of these benchmarks from a simple hosting Web page.

JavaScript Menu makes extensive use of `document.write` to build menus, with several of these calls used to register event handlers, as shown below (event handler registrations are shown in bold). Each `document.write` call causes the transaction to suspend and pass control to the iblock. The iblock uses `document.parse` to (a) parse the arguments to identify the HTML element(s) being created; (b) identify whether any event handlers are being registered and wrap them; and (c) write resulting HTML to the transaction's speculative DOM. *(2) GoogieSpell* extends the AJS library to provide a spell-checking service. When a user clicks the "check spelling" button, GoogieSpell sends an `XMLHttpRequest` to a third party server to fetch suggestions for misspelled words. We created a transactional version of GoogieSpell, whose iblock implemented a domain-specific policy that prevents an `XMLHttpRequest` once the benchmark has read domain cookies or if the target URL of

`XMLHttpRequest` does not appear on a whitelist. Such *cross-origin resource sharing* permits cross-site `XMLHttpRequest`s, and is supported by Firefox-3.5 and higher [123].

(3) Color Picker builds upon the popular jQuery library [12] and lets a user pick a color by moving sliders depicting the intensities of red, blue and green. We executed the entire benchmark (including all the supporting jQuery libraries) as a transaction and encoded an iblock that disallowed modifications to the `innerHTML` property of arbitrary `<div>` nodes.

However, for this guest, it turns out that an iblock that disallows any changes to the sensitive `innerHTML` property of *any* `<div>` element is overly restrictive. This is because Color Picker modified the `innerHTML` property of a `<div>` element that it created. We therefore loosened our policy into a history-based policy that let the benchmark change `innerHTML` properties of `<div>` elements that it created. The iblock determines whether a `<div>` element was created by the transaction by querying its write set. The relevant snippet from the iblock is shown below; the `tx` variable denotes the transaction:

```
1  var ws = tx.getWriteSet(); ...
2  if (tx.getCause().match("innerHTML")
3         && ws.checkMembership(tx.getObject(), "*")
4         && !(tx.getObject() instanceof HTMLBodyElement))
5    // perform action on behalf of untrusted code
```

5.9.2 Fault Injection and Recovery

To evaluate how Transcript can help hosts detect and debug malicious guest activity, we performed a set of fault-injection experiments on a real Web application that allows integration of untrusted guest code. We used the Bigace Web content management system [2] running on our Web server as the host, and created a Web site that mashed content from Bigace with content provided by untrusted guests (each guest was included into the mashup using the `<script>` tag). We wrote guests that emulated known attacks and studied host behavior when the host (1) directly included the guest in its protection domain; and (2) used Transcript to isolate the guest.

Our experiments show that with appropriate iblock policies, speculative execution ensured

clean recovery; neither the JavaScript heap nor the DOM of the host was affected by the misbehaving guest.

- *Misplaced event handler.* JavaScript provides a `preventDefault` method that suppresses the default action normally taken by the browser as a result of the event. For example, the default action on clicking a link is to fetch the page corresponding to the URL referenced in the link. Several sites use `preventDefault` to encode domain-specific actions instead, *e.g.,* displaying a popup when a link is clicked.

  In this experiment, we created a buggy guest that displays an advertisement within a `<div>` element. This guest mistakenly registers an `onClick` event handler that uses `preventDefault` with the `document` object instead of with the `<div>` element. The result of including this guest directly into the host's protection domain is that all hyperlinks on the Web page are rendered unresponsive. We then modified the host to isolate the guest using a policy that disallows a transaction to commit if it attempts to register an `onClick` handler with the `document` object. This prevented the advertisement from being displayed, *i.e.,* the `<div>` element containing the misbehaving guest was not even created, but otherwise allowed the host to function correctly. JavaScript reference monitors proposed in prior work can prevent the registration of the `onClick` handler, but leave the `div` element of the misbehaving guest on the host's Web page.

- *Prototype hijacking.* We implemented a prototype hijacking attack by writing a guest that set the `Array.prototype.slice` function to `null`. To illustrate the ill-effects of this attack, we modified the host to include two popular (and benign) widgets, namely Twitter [21] and AddThis [1], in addition to the malicious guest. The prototype hijacking attack prevented both the benign widgets from functioning properly.

  However, when the malicious guest is enclosed within a transaction whose iblock prevents a commit if it detects prototype hijacking attacks, the host and both benign widgets worked normally. We further inspected the transaction's write set and verified that none of the heap operations attributed to the malicious guest were actually applied to the host. Although traditional JavaScript reference monitors can detect and prevent prototype hijacking attacks by blocking further `<script>` execution, they do not allow the hosts to

cleanly recover from all heap changes.

- *Oversized advertisement.* We created a guest that displayed an interactive JavaScript advertisement within a `<div>` element. In an unprotected host, this advertisement expands to occupy the full screen on a `mouseover` event, *i.e.,* the guest registered a misbehaving event-handler that modifies the size of the `<div>`. We modified the host to isolate this guest using a transaction and an iblock that prevents a `commit` if the size of the `<div>` element increased beyond a pre-specified limit. With this policy, we observed that the host could successfully prevent the undesired `<div>` modification by discarding the speculative DOM and JavaScript heap changes made by the event handler executing within the transaction.

5.9.3 Performance

We measured the overhead imposed by Transcript using both guest benchmarks, to estimate the overall cost of using transactions, and microbenchmarks, to understand the impact on specific JavaScript operations.

Guest Benchmarks

To evaluate the overall performance impact of Transcript, we measured the increase in the load time of each guest benchmark. Recall that each benchmark is included in the Web page using a set of `<script>` tags; the version that uses Transcript executes the corresponding JavaScript code within a single transaction using modified `<script>` tags. The `onload` event fires at the end of the document loading process, *i.e.,* when all scripts have completed execution. We therefore measured the time elapsed from the moment the page is loaded in the browser to the firing of the `onload` event.

To separately assess the impact of speculatively executing JavaScript and DOM operations, each experiment involved executing the benchmarks on two separate variants of Transcript, namely Transcript (full), which supports both speculative DOM and JavaScript operations, and Transcript (JS only), which only supports speculative JavaScript operations (and therefore does not isolate DOM operations of the guest). Figure 5.11 presents the results averaged over 25 runs

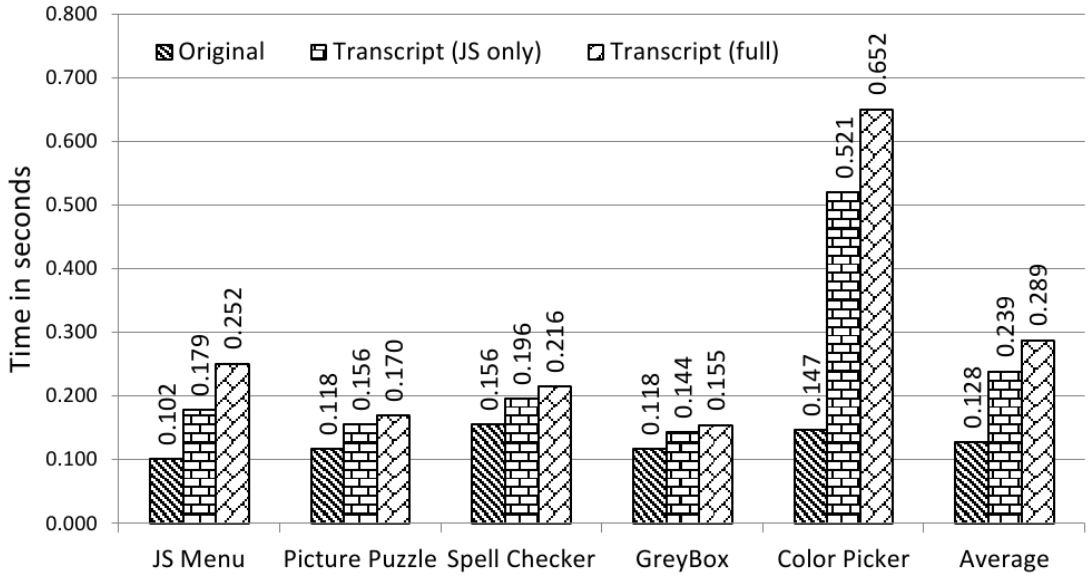


Figure 5.11: **Performance of guest benchmarks isolated using Transcript.** This chart compares the time to load the unmodified version of each guest benchmark against the time to load the transactional version in the two variants of Transcript.

of this experiment. On average, Transcript (JS only) increased load time by just 0.11 seconds while Transcript (full) increased the load time by 0.16 seconds. These overheads are typically imperceptible to end users. Only Color Picker had above-average overheads. This was because (a) the guest heavily interacted with the DOM, causing frequent suspension of its transaction; and (b) the guest had several `Array` operations that referenced the `length` of the array. Each such operation triggered a traversal of read/write sets to calculate the array length.

Microbenchmarks

We further dissected the performance of Transcript using microbenchmarks designed to stress specific functionalities. We used two sets of microbenchmarks: function calls and event dispatchers. In our experiments, we executed each microbenchmark within a transaction whose iblock simply permitted all actions and resumed the transaction without enforcing additional security policies, and compared its performance against the non-transactional version.

	Microbenchmark	Overhead
	Native Functions	
1	`eval("1")`	6.69×
2	`eval("if (true)true;false")`	6.87×
3	`fn.call(this, i)`	1.89×
	External operations	
4	`getElementById("checkbox")`	6.78×
5	`getElementsByTagName("input")`	6.89×
6	`createElement("div")`	3.69×
7	`createEvent("MouseEvents")`	3.82×
8	`addEventListener("click", clk, false)`	26.51×
9	`dispatchEvent(evt)`	1.20×
10	`document.write("<b>Hi</b>")`	1.26×
11	`document.write("<script>x=1;</script>")`	2.01×

Table 5.3: **Performance of function call microbenchmarks isolated using Transcript.**

	Event name	Overhead	
		Normalized	**Raw (μs)**
1	Drag Event (`drag`)	1.71×	97
2	Keyboard Event (`keypress`)	1.16×	150
3	Message Event (`message`)	1.17×	85
4	Mouse Event (`click`)	1.54×	86
5	Mouse Event (`mouseover`)	2.05×	88
6	Mutation Event (`DOMAttrModified`)	2.14×	88
7	UI Event (`overflow`)	1.97×	61

Table 5.4: **Performance of event dispatch microbenchmarks isolated using Transcript.**

Function Calls

We devised a set of microbenchmarks (Table 5.3) that stress the performance of Transcript's function call-handling code. Each benchmark invoked the code in first column of Table 5.3 $10,000$ times.

Recall that Transcript suspends on function calls that cause external operations and for certain native function calls, such as `eval`. Each suspend operation requires Transcript to save the state of the transaction, execute the iblock, and restore the transaction state upon the execution of a `resume` call. Most of the benchmarks in Table 5.3 trigger a suspension, which induces significant overheads. In particular, `addEventListener` had an overhead of $26.51\times$. The bulk of the overhead was induced by code in the iblock that generates wrappers for the event handler registered using `addEventListener`.

User Events

A JavaScript application executing within a transaction may dispatch user events, such as mouse clicks and key presses, which must be processed by the event handler associated with the relevant DOM node. The promptness with which events are dispatched typically affects end-user experience.

To measure the impact of transactions on this aspect of browser performance, we devised a set of microbenchmarks that dispatched user events such as clicking a checkbox, moving the mouse, pressing keys, etc. and measured the delay in handling them (Table 5.4).

In each case, code that generated and dispatched the event executed as a transaction with an iblock that allowed all actions. To measure overhead, we executed this code $1{,}000$ times and compared its performance against a native event dispatcher.

Table 5.4 presents the results, which show the normalized overhead as well as the raw delay to process a single event. As this figure shows, although the normalized overheads range from 16% to 114%, the raw delays average about 94 microseconds, which is imperceptible to end users.

SunSpider

Finally, we also tested Transcript with the SunSpider JavaScript benchmark suite by executing each of its benchmarks within a transaction. This benchmark suite reported an average overhead of $3.94\times$ across all benchmarks. In particular, we observed high overheads for benchmarks that had tight loops operating over many array elements. The overhead primarily stems from having to consult the write set for every read operation and updating the read set itself even though the iblock's permissive security policy did not consult read/write sets.

5.9.4 Complexity of Policies

To study the complexity of writing policies in Transcript, we compared the number of lines of code needed to write policies in Transcript and in Conscript [111]. We considered the policies discussed in Conscript and wrote equivalent policies in Transcript. The details of these equivalent Transcript policies are included in Appendix A. Table 5.5 compares the source lines of code

Policy	T-LOC	C-LOC	Policy	T-LOC	C-LOC
Conscript-#1	7	2	Conscript-#2	5	6
Conscript-#3	6	3	Conscript-#4	9	7
Conscript-#5	9	9	Conscript-#6	5	8
Conscript-#7	7	5	Conscript-#8	5	6
Conscript-#10	9	16	Conscript-#11	12	17
Conscript-#12	5	4	Conscript-#13	4	6
Conscript-#14	3	5	Conscript-#15	6	7
Conscript-#16	6	4	Conscript-#17	7	5

Table 5.5: **Comparing effort to write security policies in Transcript and Conscript.** This table lists the effort in lines of code required to write iblock policies in Transcript (T-LOC) and their corresponding Conscript (C-LOC) policies. Policies are numbered as in Conscript [111]. We omitted Conscript-#9 since it is IE-specific.

(counting number of semi-colons) of policies in Transcript and Conscript. This comparison shows that the programming effort required to encode policies in both systems is comparable.

5.10 Related Work

There is much prior work in the broad area of isolating untrusted guests. Transcript is unique because it allows hosts to recover cleanly and easily from the effects of malicious or buggy guests (Table 5.6). In exchange for requiring no modification to the guest, Transcript requires modifications both to the host (*i.e.,* the server side) and to the browser (*i.e.,* the client side) to enhance the JavaScript language.

5.10.1 Static Analysis

Several dynamic constructs in the JavaScript language, such as `eval`, `with` and `this`, make it intractable for static code inspection. Thus, several projects, such as AdSafe [49] and FBJS [72], have advocated the use of subsets of the JavaScript language to make it amenable for static analysis. However, safe subsets of JavaScript are non-trivial to design [75, 106, 108, 109], and also restricts code developers from using arbitrary constructs of the language in their applications. ADsafety [138] proposes a lightweight and efficient verification for JavaScript sandboxes, and has been successfully applied to ADsafe.

Despite the dynamic nature of JavaScript, there have been a few efforts at statically analyzing JavaScript code. Gatekeeper [85] presents a static analysis to validate widgets written in a subset of JavaScript. It does so by matching widget source code against a database of patterns denoting unsafe programming practices. Beacon [96] is a specialized program analysis tool targeted towards Mozilla's Jetpack framework. Beacon leverages Gatekeeper's points-To relations to determine capability leaks in Jetpack extensions. Actarus [86] is another static analysis based system that studies insecure flows in JavaScript Web applications. Its set of sources and sinks are thus based on rules targeting specific vulnerabilities. For example, the DOM property `innerHTML` or the method `document.write` is a sink because they facilitate code injection attacks. ENCAP [154] implements a flow- and context-insensitive static analysis of JavaScript to detect API circumvention.

Guha *et al.* [88] developed static techniques to improve AJAX security. Their work uses static analysis to enhance a server-side proxy with models of AJAX computation on the client. The proxy then ensures that AJAX requests from the client conform to these models. Chugh *et al.* [44] developed a staged information flow tracking framework for JavaScript to protect hosts from untrusted guests. Its static analysis identifies constraints on host variables that can be read or written by guests. The analysis validates these constraints on code loaded at runtime via `eval` or `<script>` tags, and rejects the code if it violates these constraints.

Content Security Policy (CSP) [150] implements a declarative policy that the browser must enforce on the entire application. CSP is a useful tool in preventing content injection attacks in Web applications. It does do by declaring CSP properties that state the set of trusted servers for content on the Web page. However, this places severe restrictions on how Web application pages can be structured. In comparison, Transcript places not restrictions on the content and is compatible with all legacy Web applications.

All the above mentioned static analyses are useful. However, they cannot entirely obviate the need for dynamic mechanisms to analyze JavaScript and thus, they complement Sabre.

System	Recovery	Unrestricted guest	Unmodified browser	Policy coverage
Transcript	✓	✓	✗	Heap + DOM
Conscript [111]	✗	✓	✗	Heap + DOM
AdJail [103]	✗	✓	✓	DOM(1)
Caja [119]	✗	✗	✓	Heap + DOM
Wrappers [107, 108, 110]	✗	✓(2)	✓	Heap + DOM
Info. flow [44]	✗	✓	✓	Heap
IRMs [135, 142, 176]	✗	✓	✓	Heap + DOM
Subsetting [49, 72, 108]	✗	✗	✓	Static policies(3)

Table 5.6: **Comparison of techniques to confine untrusted third party JavaScript code.** (1) Adjail uses a separate `<iframe>` to disallows guests from executing in the host's context. (2) Some wrapper-based solutions [107] restrict JavaScript constructs allowed in guests. (3) Subsetting is a static technique and its policies are not enforced at runtime.

5.10.2 Runtime Protection

Recent work on sandboxing JavaScript has traditionally focused on the use of existing browser primitives to confine untrusted third party code that may be included in Web pages as libraries, widgets and advertisements. We now list a few of the popular techniques and compare them with Transcript.

(i) Wrapper and Capability-based Sandboxing

Object capability and wrapper-based solutions (*e.g.,* [15, 107, 108, 110, 119]) create wrapped versions of JavaScript objects to be protected, and ensure that they can only be accessed by code that has the capabilities to do so. In contrast to these techniques, which provide isolation by wrapping the host's objects, Transcript wraps guest code using transactions, and mediates its actions with the host via iblocks. Prior research has also developed solutions to inline runtime checks into untrusted guests. These include BrowserShield [142], CoreScript [176], and the work of Phung *et al.* [135]. Unlike these works, Transcript simply wraps untrusted code in a transaction, and does not modify it. These works also do not explicitly address recovery.

Aspect-oriented programming (AOP) techniques have previously been used to enforce cross-cutting security policies [36, 69, 71]. Among the AOP-based frameworks for JavaScript [111, 164], Transcript is most closely related to Conscript [111], which uses runtime aspect-weaving to enforce policies on untrusted guests. Both Conscript and Transcript

require changes to the browser to support their policy enforcement mechanisms. However, unlike Transcript, Conscript does not address recovery from malicious guests, and also requires guests to be written in a subset of JavaScript. While recovery may also be possible in hosts that use Conscript, the hosts would have to encode these recovery policies explicitly. In contrast, hosts that use Transcript can simply discard the speculative changes made by a policy-violating guest.

(ii) Isolation Using `<iframe>`s

Most schemes to isolate untrusted JavaScript combine the browser's sandboxing features – Same Origin Policy and and `<iframe>`s. For example, SMash [54], Subspace [93], OMOS [177] and AdJail [103] isolate untrusted scripts by running them in `<iframe>`s served from different origins and use `postMessage` for inter-frame communication.

AdJail, in particular, aims to protect hosts from malicious advertisements [103]. It confines advertisements by executing them in a separate `<iframe>`, and uses `postMessage` to allow the `<iframe>` to communicate with the host. Hosts use access control policies to determine the set of DOM modifications allowed by an advertisement. AdJail is effective at confining advertisements, which cannot affect the host's heap. However, it is unclear whether this approach will work in scenarios where hosts and guests need to interact extensively, *e.g.,* in the case where the guest is a library that the host wishes to use.

More recently, Treehouse [91] and [30], advocate the use of *temporary* origins to leverage the isolation provided by the combination of same-origin policy and `<iframe>`s. Treehouse additionally uses isolated Web workers with a virtual DOM implementation to interpose on all DOM events, providing stronger security and resource isolation properties, but at a higher performance cost.

Blueprint [105] and Virtual Browser [40] confine guests by setting up a virtual environment for their execution. This environment is itself written in JavaScript and parses HTML and script content, thereby mediating the execution of guests on unmodified browsers. However, unlike Transcript, they do not address recovery.

Jigsaw [115] is a recent framework for isolating mashup components that leverages the

well-understood public/private keywords from object-oriented programming to make it easy for developers to tag internal data as externally visible. But, unlike Transcript that can be used with legacy Web applications, porting legacy applications to Jigsaw would require considerable effort. OMash [48] is similar to Jigsaw, and restricts communication to public interfaces declared by each page.

(iii) Browser Enhancements

Both BEEP [94] and MashupOS [161] enhance the browser with new HTML constructs. BEEP's constructs allow the browser to detect script-injection attacks, while MashupOS provides sandboxing constructs to improve the security of client-side mashups. While Transcript requires modified `<script>` tags as well, it provides the ability to speculatively execute and observe the actions of untrusted code, which neither BEEP nor MashupOS provide.

AdSentry [60] uses a shadow JavaScript engine for untrusted ad execution. The shadow JavaScript engine ensures that untrusted content will not affect the host Web content, thus protecting user privacy and the integrity of Web applications. Transcript achieves the desired effect by using speculation and the suspend/resume mechanism.

5.10.3 Using Transactions for Security

Transactions and speculative execution mechanisms have previously been used to improve software security and reliability (*e.g.,* [39, 139, 148]). However, the work most closely related to Transcript is by Sun *et al.* [152] on one-way isolation. This work describes a sandboxing mechanism that allows isolated execution of untrusted code. As in Transcript, code within the sandbox cannot modify the state of code outside, but the reverse is possible. However, their work focused on implementing such a sandbox at the granularity of operating system artifacts, such as processes and files. In contrast, Transcript discusses a similar approach but applies it to the problem of isolating JavaScript code. Accordingly, their work is realized by making changes to the operating system, whereas Transcript requires changes to the JavaScript interpreter.

5.11 Summary

This chapter shows that extending JavaScript with support for transactions allows hosting principals to speculatively execute and enforce security policies on untrusted guests. Speculative execution allows hosts to cleanly and easily recover from the effects of malicious and misbehaving guests. In building Transcript, we made several contributions, including suspend/resume for JavaScript, support for speculative DOM updates, and novel strategies to implement transactions in commodity JavaScript interpreters. All these together combine to provide complete *isolation* of untrusted third party JavaScript code.

Part II

Enhancing Web Platform Extensibility

Chapter 6

A Systems Approach to Enhance Web Platform Extensibility

Web software developers *assume* that Web browsers provide a secure and rich, extensible environment. However, in reality, the browsers present a buggy and brittle interface, which makes it hard to write secure and robust Web software in a browser-neutral manner. In this chapter, we present Atlantis, a novel, extensible Web browser that leverages exokernel and virtualization principles to improve the security, robustness and extensibility of the Web platform. We discuss its design and implementation, several heuristics to improve performance, and evaluate its effectiveness in real-world scenarios.

6.1 Problem

The Web browser exports a huge and complex API for Web applications that is hard to secure and difficult to implement correctly. Further, implementations of the various browser subsystems differ significantly across different Web browsers. Thus, a Web application's execution varies across the different Web browsers. Modern Web pages provide HTML, JavaScript and CSS to the browser in the hope that the browser would correctly layout and render the applications as intended by the developer. However, this is not guaranteed [63, 99, 128, 134], and the Web applications themselves cannot rectify or influence any stage of the page load mechanism due to the black box nature of the various browser subsystems. Developers try to circumvent the problem by using JavaScript frameworks such as jQuery [12], which provide a high-level abstraction layer to hide browser-dependent code paths. However, these frameworks cannot hide all implementation bugs in the different Web browsers. Thus, the abstraction libraries themselves may perform differently on different browsers due to unexpected incompatibilities [70, 95].

All the above issues make it difficult for Web developers to reason about the robustness and the security of their applications.

6.2 Motivating Examples

We now briefly discuss two recent examples of vulnerabilities in browser interfaces that clearly demonstrate (i) the black-box nature of browser interfaces, and (ii) the dependence of Web applications on browser vendors to issue appropriate vulnerability fixes, which leaves them vulnerable till that time.

(i) `Event` interface vulnerability in Internet Explorer. As recently as December, 2012, a new security vulnerability [9] in Internet Explorer versions 6–10 was discovered, which allows user's mouse cursor to be tracked anywhere on the screen, even if the browser window is inactive, unfocused, or minimized. The vulnerability is notable because it compromises the security of virtual keyboards and virtual keypads.

The vulnerability enables an attacker to get access to a user's mouse movements simply by displaying a malicious advertisement on any Web page the user visits. Thus, all popular sites serving third party advertisements such as YouTube or the New York Times become an attack vector. All this can happen even if the user is security conscious and never installs any untoward software.

The reason for the vulnerability is the Internet Explorer's event model, which populates the global `Event` object with attributes relating to mouse events, even in situations where it should not. Combined with the ability to trigger events manually using the `fireEvent()` method, this allows JavaScript in any Web page or in any `<iframe>` within any Web page to poll for the position of the mouse cursor anywhere on the screen and at any time, even when the tab containing the page is not active or when the Internet Explorer window is unfocused or minimized. The `fireEvent()` method also exposes the status of the control, shift, and alt keys.

(ii) Arbitrary code execution in Firefox. In versions of Mozilla Firefox before v17.0, if a user were tricked into opening a specially crafted Web page that involved setting of Cascading

Style Sheets (CSS) properties in conjunction with SVG text, the adversary could execute arbitrary code with the privileges of the user invoking the Web browser [5]. Although, memory corruption bugs are not novel, there are several CVE vulnerability disclosures involving image rendering, displaying multimedia content using buggy plugins, etc., that can lead to a full system compromise.

There are numerous other examples involving other browser subsystems that highlight the incompatibilities between different Web browsers, and [112] provides a detailed discussion on the subject. The above examples, along with those in [112], clearly demonstrate that it is extremely difficult to write secure and robust Web applications in a browser-neutral manner.

6.3 Our Approach: Virtualize the Web Application Stack

We address the issues of security and robustness of Web applications by developing a new Web browser called Atlantis, which leverages exokernel [67] principles. An exokernel enables safe and efficient multiplexing of all the raw physical resources available to the operating system by allowing applications to manage and control these resources. Atlantis's exokernel design has been guided by the insight that the modern Web browser and the Web application stack is too complicated to be implemented by any browser in a robust, secure way, such that is compatible with all Web pages. Thus, the Atlantis exokernel defines a narrow API that provides basic low-level primitives for rendering, network connectivity, and execution of abstract syntax trees that respect the browser's same-origin security policies. The Atlantis kernel places few restrictions and each Web page can compose the low-level primitives to define a richer, more secure and robust, high-level application runtime that is controlled by the Web page itself.

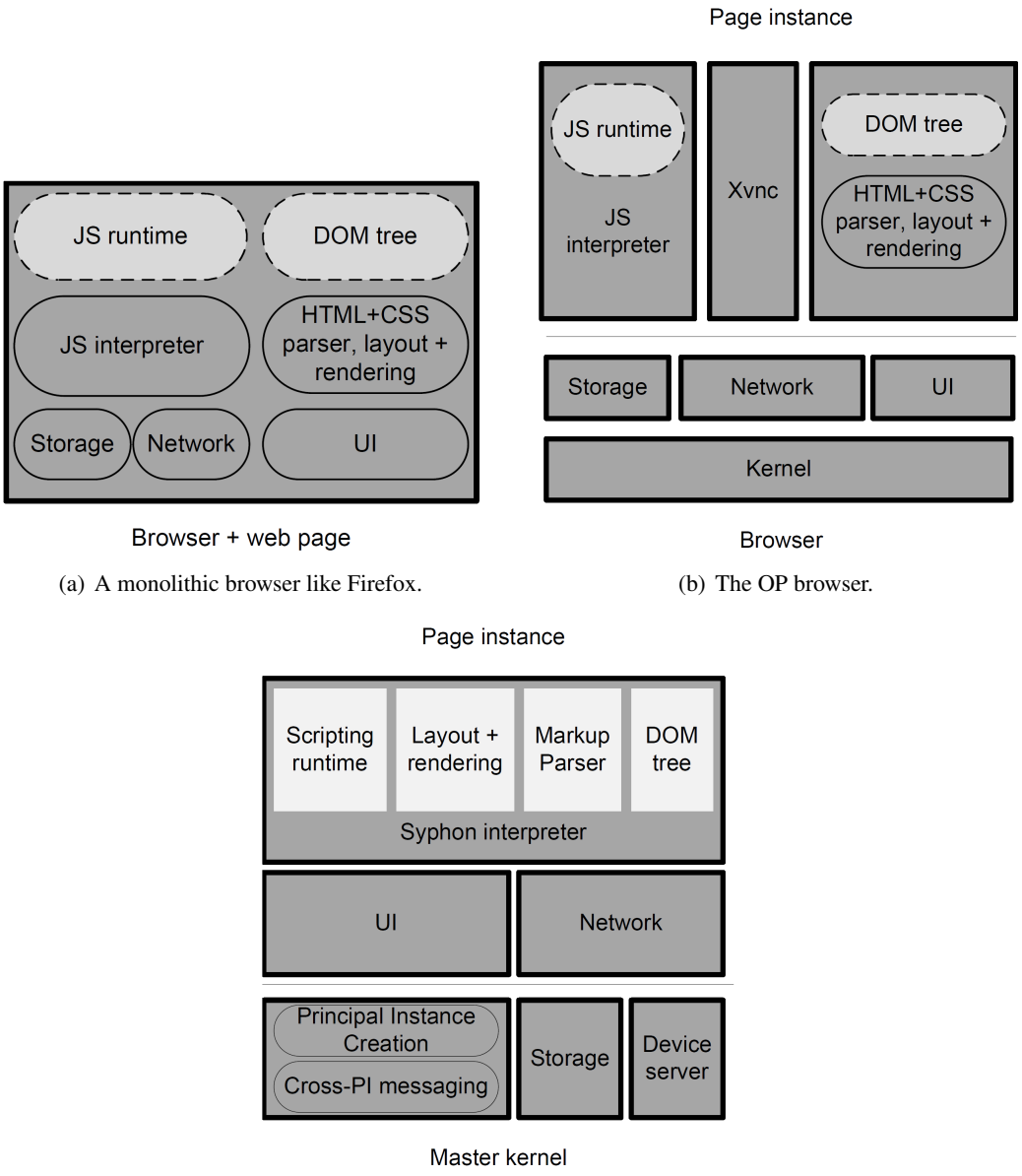
Atlantis provides Web developers with the ability to customize the runtime environment for their pages. Thus, developers can define and use any scripting or markup technologies for their Web applications. However, we envision that in most cases the developers will utilize a third party or open-source implementation of the Web application stack written in JavaScript and compiled to Atlantis ASTs. Atlantis is agnostic about the application's Web stack, and its main role is to enforce the same origin policy and provide fair allocation of low-level system resources.

Most legacy Web browsers, like Firefox, Chrome or Internet Explorer, and even recent microkernel browsers like OP [83, 84] and IBOS [155], are tightly coupled to the rigid browser abstractions such as rendering engines and parsers. For example, OP isolates the JavaScript interpreter and the HTML renderer using a process-level isolation mechanism connected by a message passing interface. However, all these abstractions are still managed by native code that a Web page cannot introspect, extend, or modify in a principled way. In contrast, Atlantis leverages exokernel [67] principles to allow Web applications to manage their own complexity, which not only provides a more *secure*, but also a more *extensible* execution environment. This extensibility by enabling complete independence from opaque, black-box interfaces of browser subsystems also allows Web pages to be more *robust* in nature.

Since Atlantis enables Web pages to customize their own execution environments, vulnerabilities such as those described in Section 6.2 can be trivially fixed. For example, Web applications could themselves fix the implementation of the `Event` object to ensure that its properties cannot be accessed by JavaScript when the window is not in focus, or even in the special case when virtual keyboards are used. Vulnerabilities involving memory corruption can also be prevented by fixing the appropriate implementations in the Web application's runtime. As will be described in Section 6.4, Atlantis's isolation of browser subsystems ensures that memory corruption of one subsystem does not affect the other parts of the Web browser. In each case, the Web application developers and the users of the Web application are not dependent on the browser vendors to issue a fix for the vulnerabilities. In Section 6.8.1, we further demonstrate the ease of extending the Web stack for providing enhanced security and robustness. We show that it is trivial to modify the DOM tree `innerHTML` feature so that a sanitizer [119] is automatically invoked on write accesses. This allows a Web page to prevent script injection attacks [126].

6.4 Isolating Browser Components

We now describe the architecture of a modern Web browser that addresses the core problem of enabling development of secure and robust Web applications in a browser-neutral manner. We also explain why both legacy and new research browsers fail to address the problem.



(a) A monolithic browser like Firefox.

(b) The OP browser.

(c) Atlantis.

Figure 6.1: **Browser architectures.** Rectangles represent strong isolation containers (either processes or C# `AppDomains`). Rounded rectangles represent modules within the same container. Solid borders indicate a lack of extensibility. Dotted borders indicate partial extensibility, and no border indicates complete extensibility.

Figure 6.1(a) shows the architecture of a monolithic legacy browser like Firefox or Internet Explorer. Monolithic browsers share two important characteristics. First, a browser "instance" consists of a process containing all of the components mentioned in Section 2.2. In some monolithic browsers, separate tabs receive separate processes; however, within a tab, browser

components are not isolated.

The second characteristic of a monolithic browser is that, from the Web page's perspective, all of the browser components are either black box or grey box. In particular, the HTML/CSS parser, layout engine, and renderer are all black boxes, and the application cannot monitor or directly influence the operation of these components. Instead, the application provides HTML and CSS as inputs, and receives a DOM tree and a screen repaint as outputs. The JavaScript runtime is grey box, since the JavaScript language provides powerful facilities for reflection and dynamic object modification. However, many important data structures are defined by native objects, and the JavaScript proxies for these objects are only partially compatible with JavaScriptś object semantics. The reason is that these proxies are bound to browser state that is hidden from an application. Thus, seemingly innocuous interactions with native code proxies, like extending `prototypes` on DOM objects, may force internal browser structures into inconsistent states [113].

Figure 6.1(b) shows the architecture of the OP microkernel browser [83]. The core browser consists of a network stack, a storage system, and a user-interface system. Each component is isolated in a separate process, and they communicate with each other by exchanging messages through the kernel. A *Web page instance* runs atop these core components. Each instance consists of an HTML parser/renderer, a JavaScript interpreter, an Xvnc [156] server, and zero or more plugins. All of these are isolated in separate processes and communicate via message passing. For example, the JavaScript interpreter sends messages to the HTML parser to dynamically update a page's content; the parser sends screen updates to the Xvnc server, which forwards them to the UI component using the VNC protocol [156]. The kernel determines which plugins to load by inspecting the MIME types of HTTP fetches (e.g., `application/x-shockwave-flash`). The kernel loads each plugin in a separate process, and the plugins use message passing to update the display or the page's HTML content. IBOS [155] is another microkernel browser that uses a similar isolation scheme.

Although, OP and IBOS provide better security and fault isolation than monolithic browsers, both OP and IBOS use standard, off-the-shelf browser modules to provide the DOM tree, the JavaScript runtime, the layout and the rendering engine. Thus, these browsers still

present Web developers with opaque interfaces similar to legacy Web browsers.

6.5 Atlantis Design

Figure 6.1(c) depicts Atlantis's architecture, which is inspired by exokernel [67] principles. At its core, Atlantis has a *m*aster kernel that contains a switchboard process, a device server, and a storage manager. The switchboard creates isolated instances for Web pages, and manages message passing between these instances and other subsystems. The device server arbitrates hardware access to devices like Web cameras and microphones. The storage manager provides a key/value interface for managing persistent data.

The storage space is partitioned into a single public area and multiple, private, per-domain areas, where each "domain" is defined by the tuple `<protocol, host name, port>`. Data from the public area can be read or written by any domain, but the storage manager authenticates all requests to private data for each domain. When the switchboard creates a fresh instance for a domain X, it assigns an authentication token to X and sends a message to the storage manager to bind the token to X. Later, when X wishes to access private storage, it must include its authentication token in the request. Section 6.5.2 explains the usefulness of unauthenticated public storage.

Atlantis creates a separate *principal instance* for each Web domain. For example, if a user opens two separate tabs for the same URL, say `http://a.com/foo.html`, then Atlantis will create two separate principal instances with each containing a *per-instance Atlantis kernel* and a *script interpreter*. The instance kernel contains the UI and the network modules. This creation of separate principal instances for each Web domain effectively multiplexes the browser's available resources, similar to how an exokernel operating system manages the available resources across different user applications.

The network manager implements protocols like `http://` and `file://`, while the UI manager creates a new C# `Form` and registers handlers for low-level GUI events on that form. The UI manager also forwards these events to the application-defined runtime, and updates

the `Form`'s bitmap in response to messages from the page's layout engine. The script interpreter executes abstract syntax trees (ASTs) that encode a new language called Syphon (Section 6.5.3). A Web page installs a custom HTML/CSS parser, DOM tree, layout engine, and high-level script runtime by compiling the runtime environment to Syphon ASTs and executing it. The Syphon interpreter implements the browser's same-origin policy. However, it is completely agnostic about the nature of the application defined execution environment. Thus, unlike in current browsers, the application defined Web stack has no dependencies on internal browser state.

Atlantis strongly isolates each principal instance's network stack, UI manager, and Syphon interpreter to run in separate native threads. Thus, these components can run in parallel and take advantage of multicore processors. Although these threads reside within a single process, they are strongly isolated from each other using C# `AppDomains` [31]. These `AppDomains` rely on the `.NET` runtime to enforce memory safety and protection *within* a single process. Code executing within an `AppDomain` cannot directly access memory outside its domain. However, an `AppDomain` can explicitly expose entry points that are accessible by other `AppDomain`.

Figure 6.1(c) shows the different modules in a Web application's execution runtime that execute within the `AppDomain` of the interpreter. However, Syphon provides several language primitives that enables isolation of from each other. For example, an application can partition its Syphon code into privileged and unprivileged components, such that only privileged code can make kernel calls. Section 6.5.3 provides a detailed discussion of Syphon's protection features. An application can use several of these isolation features to protect itself from itself – the Syphon interpreter is agnostic to the meaning of the protection domains that it enforces, and Atlantis's security guarantees do not depend on applications using Syphon's protection mechanisms.

6.5.1 Initializing a New Principal Instance

When the master kernel instantiates a new instance kernel, it provides the instance kernel with a storage authentication token, which then initializes its UI manager, network stack, and Syphon interpreter. Next, the instance kernel fetches the markup associated with its

```
<environment>
  <compiler='http://a.com/compiler.syp'>
  <markupParser='http://b.com/parser.js'>
  <runtime='http://c.com/runtime.js'>
</environment>
```

Figure 6.2: **A Web application can redefine its runtime using an `<environment>` tag at the top of its markup.**

page's URL. Atlantis is completely agnostic about whether this markup is HTML or something else. However, Web applications can redefine their execution runtime by including a special `<environment>` tag in the beginning of their markup. Figure 6.2 shows an example. The `environment` tag contains at most three elements:

- The `<compiler>` must provide code to transform the application defined scripting runtime into Syphon ASTs. The compiler itself must already be compiled to Syphon. If no compiler is specified, Atlantis assumes that the page's runtime environment is directly expressed in Syphon.

- The `<markupParser>` specifies the code that the application will use to parse itself following the end of the `<environment>` tag.

- The `<runtime>` provides the rest of the execution environment, e.g., the layout engine, the DOM tree, and the high-level scripting runtime.

The compiler defines `compiler.compile(srcString)` as an entry point method that takes a string of application-specific script code as input, and outputs the equivalent Syphon code. The instance kernel invokes `compiler.compile()` to generate executable code for the markup parser and the runtime library. After installing this code, the kernel passes the application's markup to the parser's entry point method – `markup.parse(markupStr)`. At this point, the Atlantis kernel relinquishes control to the application, which parses its markup and invokes the kernel to fetch additional objects, update the screen, etc.

If the instance kernel does not find an `<environment>` tag in the page's markup, it assumes that the page wishes to execute atop the traditional Web stack. In this case, Atlantis loads

createPI(url, width, height, topX, topY, isFrame=false)	Create a new principal instance. If isFrame is true, the new instance is the child of a parent frame. Otherwise, the new instance is placed in a new tab.
registerGUICallback(dispatchFunc)	Register an application-defined callback which the kernel will invoke when GUI events are generated.
renderImage(pixelData, width, height, topX, topY, options) **renderText**(textStr, width, height, topX, topY, options) **renderGUIwidget**(widgetType, options)	The application's layout engine uses these calls to update the screen. Strictly speaking, renderImage() is sufficient to implement a GUI. However, Web pages that want to mimic the native look-and-feel of desktop applications can use native fonts and GUI widgets using renderText() and renderWidget().
HTTPStream **openConnection**(url)	Open an HTTP connection to the given domain. Returns an object supporting blocking writes and both blocking and non-blocking reads.
sendToFrame(targetFrameUrl, msg)	Send a message to another frame. Used to implement cross-frame communication like postMessage().
executeSyphonCode(ASTsourceCode)	Tell the interpreter to execute the given AST.
persistentStore(mimeType, key, value, isPublic, token) string **persistentFetch**(mimeType, key, isPublic, token)	Access methods for persistent storage. The storage is partitioned into a single public area, and multiple, private, per-domain areas. The token argument is the authentication nonce created by the switchboard.

Table 6.1: **Primary kernel APIs in the Atlantis Web browser.**

its own implementation of the HTML/CSS/JavaScript environment. From the page's perspective, this stack behaves like the traditional stack, with the important exception that everything is written in pure JavaScript and with no dependencies on the internal browser state. Thus, modifying DOM prototypes will work as expected [92, 113], and placing getters or setters on DOM objects will not break event propagation [113]. Of course, Atlantis's default Web stack might have bugs, *but the application can fix these bugs itself without the fear of breaking the browser*.

6.5.2 The Kernel Interface

As the Web application executes, it interacts with its instance kernel using the APIs in Figure 6.1. We briefly describe few of the salient aspects below.

The Web application can create a new frame or tab by invoking the createPI() kernel

call. If the new principal instance is a child frame, the instance kernel in the parent registers the parent-child relationship with the master kernel. Later, if the user moves or resizes the window containing the parent frame, the master kernel notifies the instance kernels in the descendant frames, allowing Atlantis to maintain the visual relationships between parents and children.

Communication across two frames is made possible using the `sendToFrame()` kernel call. An application can implement JavaScript's `postMessage()` as a wrapper around `sendToFrame()`. The application can also use `sendToFrame()` to support cross-frame namespace abstractions. For example, in the traditional Web stack, if a child frame and a parent frame are in the same domain, they can reference each other's JavaScript state through objects like `window.parent` and `window.frames[childId]`. An Atlantis DOM implementation supports these abstractions by interposing on these object accesses and silently generating `postMessage()` RPCs to access remote variables. Section 6.5.3 describes how Syphon supports such features.

Atlantis exports a simple key/value interface to access its persistent store, which is split into a single public space and multiple, private, per-domain areas. Accessing private areas requires an authentication token, while accessing public areas does not. Web applications can use the `persistentStore()` and `persistentFetch()` APIs to access the public storage area and private, per-domain storage, and also to implement abstractions like cookies and DOM storage [159].

Atlantis implements the Web browser cache in the public area, with cached items keyed by their URL. However, only instance kernels can write to public storage using URL keys. This ensures that when the network stack is handling a fetch for an object, it can trust any cached data that it finds for that object. The public storage area is useful for implementing asynchronous cross-domain message queues. Specifically, the public storage allows two domains to communicate without forcing them to use `postMessage()` (which only works when both domains have simultaneously active frames). Atlantis does not enforce mandatory access controls for the public storage volume, so domains that require confidentiality and integrity for public data must leverage security protocols atop Atlantis's storage stack.

6.5.3 Syphon: Atlantis ASTs

The Atlantis Web browser encodes a new scripting language *Syphon* that is a superset of the ECMAScript specification [62], which is the standard that most browser vendors follow for implementing JavaScript in Web browsers. Atlantis Web applications pass abstract syntax trees (ASTs) to the Syphon interpreter for execution instead low-level bytecodes, as in Java applets. There are two main reasons for this design choice. First, ASTs are easier to optimize than bytecodes, because they retain semantic relationships required for optimizations. Bytecodes often obscure any semantic relationships between objects before optimizations can take place [151]. Second, ASTs can trivially recreate source code while it is difficult to reconstruct source code from bytecodes. This feature is especially useful when debugging an application consisting of scripts from multiple authors.

Syphon has a generic tree syntax that is amenable to serving as a compilation target for higher-level languages that may or may not resemble JavaScript. In this section, we describe some key features of the Syphon language that ease the construction of robust, application-defined execution runtime.

Object shimming: JavaScript allows Web applications to mediate reads and writes to all object properties using the *getter* and *setter* functions. While Syphon supports these *getter* and *setter* functions, it also introduces special *watcher* functions that execute when any property on an object is accessed or modified, including attempted deletions.

Watchers are extremely powerful, and Atlantis's default Web stack leverages them extensively. Section 6.8.2 describes how watchers enable Atlantis's default Web stack to invoke input sanitizers whenever an untrusted user input modifies any sensitive runtime variables. Watchers are also useful for implementing cross-frame namespace accesses, like `window.parent.objInParent`. To do so, the runtime stack defines a watcher on the `window.parent` object and resolves property accesses by issuing `sendToFrame()` calls to a namespace server in the parent frame.

Method binding and privileged execution: Typically, an Atlantis application consist of low-level code, like the layout engine and the scripting runtime, and higher-level application code that does not directly invoke the Atlantis kernel APIs. Thus, Syphon provides several

language features that allow applications to isolate the low-level code from the high-level code, effectively creating an application-level kernel. Like in JavaScript, a Syphon method can be assigned to an arbitrary object and invoked as a method of that object. However, Syphon supports the binding of a method to a specific object, thereby preventing the method from being invoked with an arbitrary `this` reference.

Syphon also supports the notion of *privileged execution*. Syphon code can only invoke kernel calls if the code belongs to a privileged method, i.e., it has a privileged `this` reference. By default, Syphon creates all objects and functions as privileged. However, an application can call Syphon's `disableDefaultPriv()` function to turn off that behavior. Subsequently, only privileged execution contexts will be able to create new privileged objects.

Atlantis's default Web stack leverages the above mentioned features to prevent higher-level application code from arbitrarily invoking the kernel APIs or modifying critical data structures in the DOM tree. However, the Atlantis kernel is agnostic as to whether the application takes advantage of features like privileged execution. These features have no impact on Atlantis's security guarantees, and they merely help a Web application to protect itself from untrusted third party JavaScript code.

Strong typing: By default, Syphon variables are untyped, as in JavaScript. However, Syphon allows programs to bind variables to types, facilitating optimizations in which the script engine generates fast, type-specific code instead of slow, dynamic-dispatch code for handling generic objects.

Note that strong primitive types have straightforward semantics, but the meaning of a strong object type is unclear in a dynamic, prototype-based language in which object properties, including prototype references, may be fluid. To better support strong object types, Syphon supports ECMAScript v5 notions of *object freezing*. By default, objects are `Unfrozen`, meaning that their property list can change in arbitrary ways at runtime. An object which is `PropertyListFrozen` cannot have old properties deleted or new ones added. A `FullFreeze` object has a frozen property list, and all of its properties are read-only. An object's freeze status can change dynamically, but only in the stricter direction. By combining strong primitive typing with object freezing along a prototype chain, Syphon can simulate

traditional classes in strongly typed languages.

Threading: Syphon supports a full threading model with locks and signaling. Syphon threads are more powerful than HTML5 Web workers [165] for two reasons. First, Web workers cannot access or modify DOM objects in the native code because this could interfere with the browser's internal state. In contrast, Syphon DOM trees reside in application-layer code, and thus, an application can define a multi-threaded layout engine without the possibility of corrupting internal browser state. Second, Web workers can only communicate via asynchronous messages, which requires the browsers to handle serialization and deserialization of objects across thread boundaries. In contrast, Syphon threads avoid this overhead since they are just thin wrappers around native OS threads.

6.5.4 Hardware Access

Traditionally, JavaScript has lacked access to hardware devices, such as Web cameras and microphones. Thus, most Web applications use content plugins such as Flash and Silverlight to access these devices. Atlantis, like Gazelle and OP, loads plugins in separate processes and restricts their behavior using same-origin checks.

The new HTML5 specification [171] allows JavaScript programs to access hardware devices through a combination of new HTML tags and JavaScript objects. For example, the `<device>` tag [170] creates a Web camera object in the JavaScript namespace, and the JavaScript interpreter translates accesses and modifications to the object's properties into specific device commands. Similarly, the `navigator.geolocation` object exposes location data gathered from GPS [169].

While HTML5 ensures that JavaScript has first-class access to hardware devices, it entrusts device security to the JavaScript interpreter of an unsandboxed browser. Interpreters are complex and often, buggy. For example, there have been several attacks on the JavaScript garbage collectors in Firefox, Safari, and Internet Explorer [51]. A compromised JavaScript interpreter in an HTML5 compatible Web browser would enable an attacker to gain full access to all of the user's hardware devices.

In contrast to HTML5, Atlantis sandboxes the Syphon interpreter, preventing it from directly accessing hardware. Instead, Web pages use the Gibraltar AJAX protocol [102] to access hardware. The master kernel uses a separate process to isolate *device server*. The device server directly accesses the hardware using native code, and exports a Web server interface on the `localhost` address. Principal instances can access the hardware by sending AJAX requests to the device server. For example, a page that wants to access a Web camera could do so by sending an AJAX request to `http://localhost/WebCam`, specifying various device commands in the HTTP headers of the request. Users authorize individual Web domains to access individual hardware devices, and the device server authenticates each hardware request by looking at its `referer` HTTP header. This header identifies the URL (and thus the domain) that issued the AJAX request. A detailed discussion on the Gibraltar device access protocol is available in [102].

In Atlantis, each principal instance runs its own copy of the Syphon interpreter in a separate `AppDomain`. Thus, even if a malicious Web page compromises its interpreter, it cannot learn which other domains have hardware access unless those domains willingly respond to `postMessage()` requests for that information. Even if domains collude in this fashion, the instance kernel implements the networking stack, so Web pages cannot fake the `referer` fields in their hardware requests unless they also subvert the instance kernel.

6.6 Implementation

We implemented the entire Atlantis browser architecture in C# and it contains 8634 lines of code, of which 4900 lines belong to the Syphon interpreter, 358 belong to the master kernel, and the remainder belong to the instance kernel and the messaging library shared by various components. We implemented the full kernel interface described in Section 6.5.2, but we have not yet ported any plugins to Atlantis. The Syphon interpreter implements all of the language features mentioned in Section 6.5.3.

Syphon interpreter: The Syphon interpreter implements all of the language features described in Section 6.5.3, like watchers, object freezing, etc. The interpreter represents each type of

Syphon object as a subclass of the `SyphonObject` C# class, where `SyphonObject` implements all non-primitive objects as dictionaries.

Much like JavaScript, Syphon also allows applications to dynamically add and remove variables from a scope, create closures i.e., functions that remember the values of nonlocal variables in the enclosing scope, etc. Syphon also manages object scope chains similar to JavaScript. Each scope in the scope chain is implemented as a dynamically modifiable dictionary. Thus, each namespace operation requires the modification of one or more hash tables.

These modifications can be prohibitively expensive, so the Syphon interpreter performs several optimizations to minimize dictionary operations. For example, to prevent the interpreter from having to scan a potentially deep scope chain for every variable access, the interpreter creates a unified cache that holds variables from various levels in the scope hierarchy. When ever a variable is accessed for the first time, the interpreter searches the scope chain, retrieves the relevant object, and places it in the unified cache.

Each function invocation requires the interpreter to allocate and initialize a new scope dictionary, while each function return requires deletion of the corresponding dictionary. The interpreter tries to avoid such costs by inlining functions, because the overall cost of invoking an inlined function is much less than the cost of invoking a non-inlined one. But to do so, the interpreter must modify function arguments and local variables, rewrite the function code to reference these modified variables, and embed these variables directly in the name cache of the caller. The function call itself is implemented as a direct branch to the rewritten function code, and when the inlined function returns, the interpreter must destroy the modified name cache entries. While the interpreter also inlines closures, it does not inline functions that generate closures, since non-trivial bookkeeping is required to properly bind closure variables.

The Syphon interpreter and the browser kernel are written in C# and compiled to CIL bytecodes. When the user invokes the browser, `.NET` just-in-time compiler dynamically translates it to x86 instructions. In our current prototype, the interpreter compiles Syphon ASTs to high-level bytecodes, and then interprets those bytecodes directly. We did write another Syphon interpreter that directly compiled ASTs to CIL, but we encountered several challenges to make

it fast. For example, to minimize the overhead of function calls in Syphon, we wanted to implement them as direct branches to the starting CIL instructions of the relevant functions. Our experiments showed that this function invocation mechanism was faster than placing the CIL for each Syphon function inside a C# function and then invoking that C# function using the standard CIL `CallVirt` instruction. Unfortunately, CIL does not support indirect branches. Thus, it was difficult to implement function returns since in Syphon, any given function can return to many different call sites. We implemented function returns by storing the call site program counters on a stack, and upon function return, using the topmost stack entry to index into a CIL switch where each case statement was a direct branch to a particular call site.

Unfortunately, this return technique, as described thus far, does not work. This is because CIL is a stack-based bytecode, and the default `.NET` JIT compiler assumes that for any instruction that is only the target of backward branches, the evaluation stack is empty immediately before that instruction executes [137]. This assumption was intended to simplify the JIT compiler, since it allows the compiler to determine the stack depth at any point in the program using a single pass through the CIL. Unfortunately, this assumption means that if a function return is a backwards branch to a call site, the CIL code after the call site must act as if the evaluation stack is initially empty; otherwise, the JIT compiler will declare the program invalid. If the evaluation stack is *not* empty before a function invocation (as is often the case), the application must manually save the stack entries to an application-defined data structure before branching to a function's first instruction. One could use forward branches to implement function returns, but then function *invocations* would have to use backwards branches, leading to a similar set of problems. Even with the overhead of manual stack management, branching function calls and returns were still faster than using the `CallVirt` instruction. However, this overhead did reduce the overall benefit of the technique, and the overhead would be avoidable with a JIT compiler that did not make the empty stack assumption.

We encountered several other challenges with the default JIT compiler. For example, we found that the JIT compiler was quick to translate the CIL bytecodes for the Atlantis interpreter that were generated statically, but it was much slower to translate the CIL bytecodes representing the Syphon application that were generated dynamically. For example, on several

macrobenchmarks, we found that the CIL-emitting interpreter was spending twice as much time in JIT compilation as the high-level bytecode interpreter, even though the additional CIL to JIT (the CIL belonging to the Syphon program) was much smaller than the CIL for the interpreter itself.

Given the above issues, our current Atlantis prototype directly interprets the high-level bytecode. However, we plan on using the SPUR framework [38] in the next version of Atlantis. The SPUR project has shown that significant performance gains can be realized by replacing the default `.NET` JIT engine with a custom one that can perform advanced optimizations like type speculation and trace-based JITing.

Default Web Stack: If a Web application does not provide its own high-level runtime, it will use Atlantis's default stack. This stack contains 5581 lines of JavaScript code which we compiled to Syphon ASTs using an ANTLR [133] tool chain. 65% of the code implements the standard DOM environment, providing a DOM tree, an event handling infrastructure, AJAX objects, etc. The remainder of the code handles markup parsing, layout calculation, and rendering. Our DOM environment is quite mature, but our parsing and layout code is the target of active development; the latter set of modules are quite complex and require clever optimizations to run quickly.

6.7 Discussion: Practical Issues with Atlantis Web Browser

Atlantis leverages exokernel [67] principles to enable each Web page to ship its own implementation of the Web stack. Each Web page can freely customize its execution environment to its specific needs, thereby reducing its dependence on the black-box browser subsystems. It is possible that individual exokernel implementations might be buggy. However, exokernel browsers are much simpler than monolithic browsers, due to a narrower browser API they export. Thus, in principle, their bugs should be smaller in number and easier to fix. Of course, an exokernel browser is only interesting when paired with a high-level runtime. Since each page chooses its own runtime that it wishes to include, it can modify that runtime as it requires. Thus, from the perspective of a Web developer, an exokernel browser seems easier to program than a monolithic browser, and reasoning about Web application security, robustness and portability

challenges is also simpler.

Even if a single exokernel interface becomes the *de facto* browser design, it is always a possibility that individual vendors will expand the narrow interface or introduce non-standard semantics for differentiating themselves. It seems impossible to prevent such feature creep. However, we believe that innovation at a low semantic level happens more slowly than innovation at a high semantic level. For example, fundamentally new file system features are created much less frequently than new application types that happen to leverage the file system. Thus, we expect that incompatibilities due to different exokernel APIs will arise much less frequently than incompatibilities between different monolithic browsers.

Exokernel browsers, like Atlantis, allow individual Web applications to define their own runtime environments, like HTML parsers, DOM implementations, layout engines, etc. Multiple implementations of each component will undoubtedly arise, and these implementations may become incompatible with each other. Furthermore, certain classes of components may be rendered unnecessary for some Web applications. For example, if an application decides to use SGML instead of HTML as its markup language, then it does not require an HTML parser. While Atlantis allows developers to use any runtime stack, in practice, most developers will not create runtimes from scratch, in the same way most Web developers today do not create their own JavaScript GUI frameworks. Instead, most Atlantis applications will use stock runtimes that are written by popular vendors or open-source efforts, and which are incorporated into applications with little or no modification. Only complex sites or those with uncommon needs, and having the technical expertise will write a heavily customized runtime.

6.8 Evaluation

In this section, we explore three issues. First, we discuss the security of the Atlantis browser with respect to various threats. Second, we demonstrate how easy it is to extend the demonstration Atlantis Web stack. Finally, we examine the performance of Atlantis on several microbenchmarks and macrobenchmarks.

6.8.1 Security

Prior work has investigated the security properties of microkernel browsers [83, 84, 155, 162]. Here, we briefly summarize these properties in the context of Atlantis, and explain why Atlantis provides stronger security guarantees than prior microkernel browsers.

Trusted computing base: Atlantis's core runtime contains 8634 lines of trusted C# code for implementing the instance kernel, the master kernel, the Syphon interpreter, and the IPC library. These modules depend on the `.NET` runtime, which is also included in Atlantis's trusted computing base. However, unlike the millions of lines of non-type safe C++ code found in Internet Explorer, Firefox, and other commodity browsers, the `.NET` runtime is type-safe and memory managed, Thus, we believe that Atlantis's threat surface is comparatively much smaller, particularly given its narrow exokernel interface.

Our Atlantis prototype also includes 5581 lines of JavaScript representing the demonstration Web stack, and an ANTLR [133] tool chain which compiles JavaScript to Syphon ASTs. These components are not part of the trusted computing base, since Atlantis does not rely on information from the high-level Web stack to guide security decisions.

Principal Isolation: Atlantis, like other microkernel browsers, strongly isolates principal instances from each other and the core browser components. This prevents a large class of attacks in monolithic browsers that place data from multiple domains in the same address space, and lack enforcement of same-origin checks [43] in a centralized manner. Atlantis uses process-level isolation for plugins and enforces the same-origin checks as experienced by other Web content to prevent the full browser compromise that results when a monolithic browser has a compromised plugin in its address space [83, 84, 162].

Atlantis has a single master kernel and multiple, sandboxed per-instance kernels. This is in contrast to Gazelle, OP, and IBOS that only use a single browser kernel, which, although memory isolated, is shared by all principal instances. If this kernel is compromised, the entire browser is compromised. For example, a subverted Gazelle kernel can inspect all messages exchanged between principal instances, tamper with persistent data belonging to an arbitrary domain, and update the visual display belonging to an arbitrary domain. In Atlantis, a subverted instance kernel can draw to its own rendering area and create new rendering areas, but it cannot

access or update the display of another instance. Similarly, a subverted instance kernel can only tamper with public persistent data (which is untrustworthy by definition) or private persistent data that belongs to the domain of the compromised principal instance. In order to tamper with resources belonging to arbitrary domains, the attacker must subvert the master kernel, which is strongly isolated from the instance kernels.

Enforcing the Same-origin Policy: The same-origin policy constrains how documents and scripts from domain `X` can interact with documents and scripts from domain `Y`. For example, JavaScript in Web pages from `X` cannot issue an `XMLHttpRequests` for JavaScript files from `Y`. This is to prevent `X`'s pages from reading `Y`'s source code. However, `X` can execute (but not inspect) `Y`'s code by dynamically creating a `<script>` tag and setting its `src` attribute to an object in `Y`'s domain. This succeeds because HTML tags are exempt from same-origin checks.

While the same-origin policy is useful, it does not prevent colluding domains from communicating. For example, if a Web page contains frames from domains `X` and `Y`, these frames cannot forcibly inspect each other's cookies, or read from or write to each other's DOM tree or JavaScript state. However, colluding domains can exchange arbitrary data across frames using the `postMessage()` API. Domains can also leak data through `iframe` URLs. For example, JavaScript executing in domain `X` can dynamically create an `iframe` with a URL pointed to domain `Y`, say `http://y.com?=PRIVATE_X_DATA`. Thus, the same-origin policy can only prevent *non-colluding* domains from tampering with each other.

As a result of Atlantis's exokernel design, it performs most, but not all of the origin checks that current browsers perform. However, Atlantis provides the same *practical* level of domain isolation. Each principal instance resides in a separate process, so each frame belonging to each origin is separated by hardware-enforced memory protection. This prevents domains from directly manipulating each other's JavaScript state or window properties. The kernel also partitions persistent storage by domain, ensuring that pages cannot inspect the cookies, DOM storage, or other private data belonging to external domains.

In Atlantis, abstractions like HTML tags and `XMLHttpRequest` objects are implemented entirely outside the kernel. Thus, when the Atlantis kernel services an `openConnection()` request, it cannot determine whether the fetch was initiated by an HTML parser upon

encountering a <script> tag, or by an XMLHttpRequest fetch. To ensure that <script> fetches work, Atlantis must also allow cross-domain XMLHttpRequest fetches of JavaScript. Although, this violates a strict interpretation of the same-origin policy, it does not change the practical security provided by Atlantis, since X's pages can trivially learn Y's JavaScript source by downloading the .js files through X's Web server. From the security perspective, it is not important to prevent the discovery of inherently public source code. Instead, it is important to protect the *user-specific client-side state* that is exposed through the browser runtime and persistent client-side storage. Atlantis protects these resources using strong memory isolation and partitioned local storage. This is the same security model provided by Gazelle [162], which assumes that principal instances will not issue cross-domain script fetches for the purposes of inspecting source code.

6.8.2 Extensibility

Atlantis allows Web pages to customize their runtime in a secure and robust manner. We produced two variants of the default Atlantis Web stack to demonstrate Atlantis's unprecedented extensibility:

(i) Safe innerHTML: Each DOM node has an innerHTML property that takes in a text string as assignment and generates the corresponding DOM tree rooted at the node itself. Web applications, such as message boards, often dynamically update themselves by taking user-submitted text and assigning it to an innerHTML property. Unfortunately, an attacker can use this vector to insert malicious scripts into the page [126]. Atlantis prevents this attack by placing a setter shim (see Section 6.5.3) on innerHTML that invokes the Caja sanitizer library [119]. Caja strips dangerous markup from the text, and the setter assigns the safe markup to innerHTML.

(ii) Stopping drive-by downloads: Assigning a URL to a frame's window.location property forces the frame to navigate to a new site. If a frame loads a malicious third party script, the script can manipulate window.location to trick users into downloading malware [46], also called drive-by downloads. We prevent this by placing a setter on window.location to allow assignments if the target URL is in a whitelist.

Implementing these extensions in Atlantis was trivial since the default DOM environment

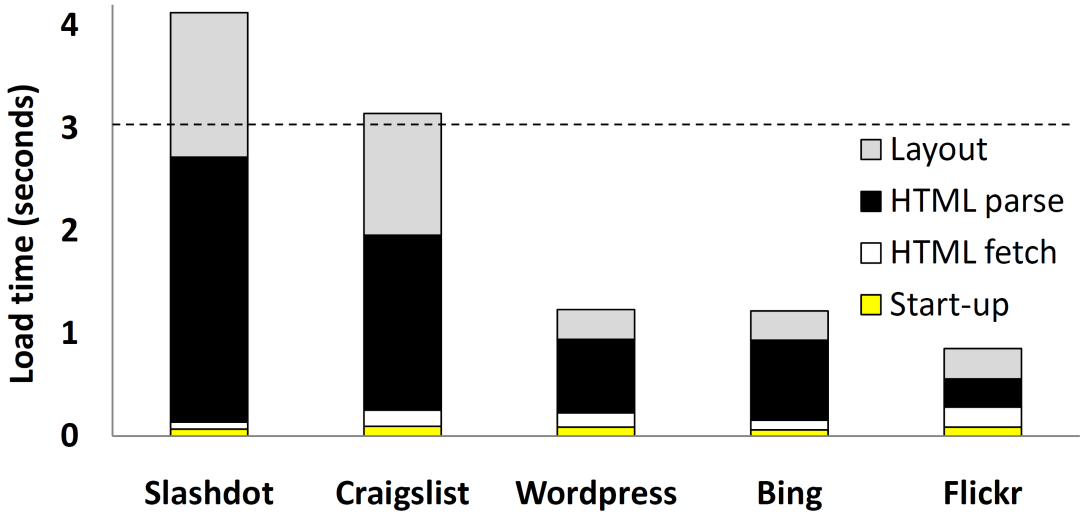


Figure 6.3: **Atlantis Web page load times.** The dotted line shows the three second window after which many users will become frustrated [77].

is written in pure JavaScript. Neither of these application-defined extensions are possible on a traditional browser. The design documents for popular browsers like Internet Explorer and Firefox explicitly forbid applications from placing setters on `window` properties. Although, placing setters on `innerHTML` is allowed, in practice it breaks the browser's JavaScript engine [113].

6.8.3 Performance

We evaluate Atlantis on several microbenchmarks and macrobenchmarks. All experiments were performed on a Lenovo Thinkpad laptop with 4 GB of RAM and a dual core 2.67 MHz processor. In our first experiment, we determine Atlantis's user perceived performance by measuring the page load time for five popular Web pages. Figure 6.3 depicts the results, where each bar represents the average of five trials and contains four components: the start-up time between the user hitting "enter" on the address bar and the kernel issuing the fetch for the page's HTML; the time needed to fetch the HTML; the time needed to parse the HTML; and the time needed to calculate the layout and render the page. The layout time includes both pure computation time and the fetch delay for external content like images. To minimize the

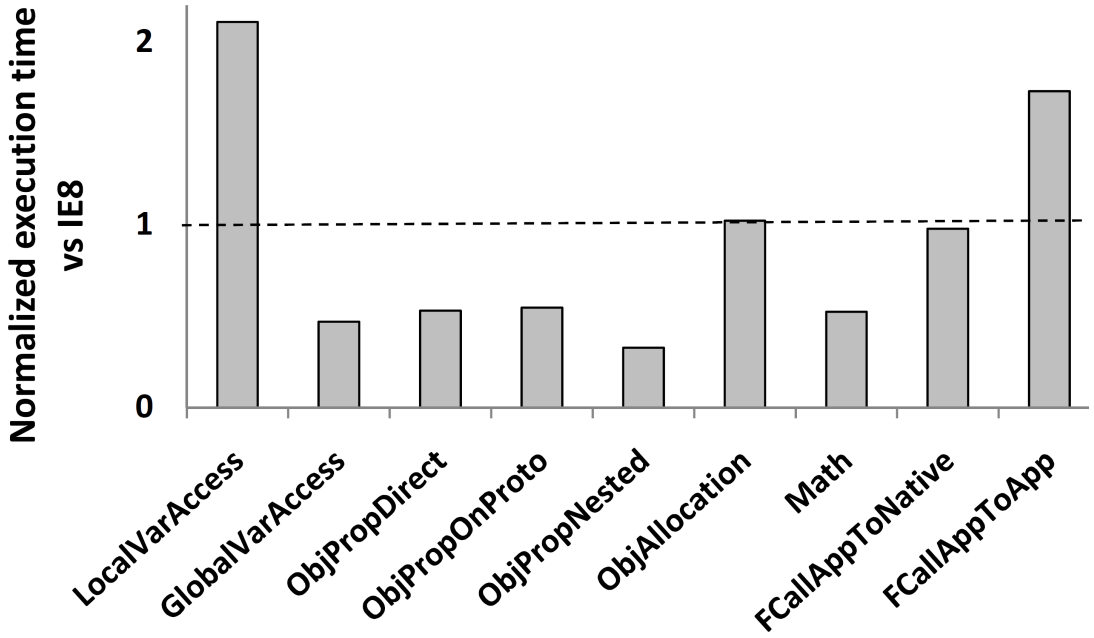


Figure 6.4: **Comparison of execution speed of Atlantis versus Internet Explorer 8 for several microbenchmarks.**

impact of network delays which Atlantis cannot control, Figure 6.3 depicts results for a warm browser cache. However, some objects were marked by their server as uncacheable and had to be refetched.

In three of the five benchmarks, Atlantis's load time is well below the three-second threshold at which users begin to get frustrated [77]. One page (Craiglist) is at the threshold, and another (Slashdot) is roughly a second over. While Atlantis's performance can be further improved, given the unoptimized nature of our prototype scripting engine, we are encouraged by the results.

Figure 6.3 shows that Atlantis load times were often dominated by HTML parsing overhead. In order to better understand this phenomenon, we evaluated several microbenchmarks as shown in Figure 6.4. Each bar represents Atlantis's relative execution speed with respect to Internet Explorer 8, and the standard deviations were less than 5% for each set of experiments. In several cases, as observed in Figure 6.4, the Syphon interpreter remains competitive

with Internet Explorer's interpreter. In particular, Syphon is two to three times faster at accessing global variables, performing mathematical operations, and accessing object properties, whether they are defined directly on an object, on an object's prototype, or on a nested object four property accesses away.

The performance penalty incurred on invoking native functions, like `String.indexOf()`, by the application code is the same on both platforms. However, Atlantis is twice as slow to access local variables, and 1.7 times as slow to invoke application-defined functions from other application-defined functions. Given that Atlantis leverages name caches for accessing both global and local variables, its relative slowness in accessing local variables is surprising. We plan to investigate this issue further. Atlantis's function invocation is slower because Atlantis performs several safety checks that Internet Explorer does not perform. These checks help to implement the Syphon language features described in Section 6.5.3. For example, the Syphon interpreter supports strongly typed variables by comparing the type metadata for function parameters with the type metadata for the arguments that the caller actually supplied. To enforce privilege constraints, the Syphon interpreter must check whether the function to invoke and its "this" pointer are both privileged. These checks make HTML parsing slow on our current Atlantis prototype, since the parsing process requires the invocation of many different functions that process strings, create new DOM nodes, and so on.

Figure 6.5 shows Atlantis's performance on several macrobenchmarks from three popular benchmark suites (SunSpider, Dromaeo, and Google's V8). We observe that Internet Explorer 8 is 1.5x to 2.8x faster than our Atlantis prototype. However, for the `OnMouseMove` program, which tracks the rate at which the browser can fire application event handlers when the user rapidly moves the cursor, Atlantis is actually about 50% faster. This is significant, since recent empirical work has demonstrated that most modern Web pages consist of a large number of small callback functions that are frequently invoked [141]. Note that firing application-defined event handlers requires native code to invoke application-defined code. The `FCallAppToNative` experiment in Figure 6.4 measures the costs for application code to call native code.

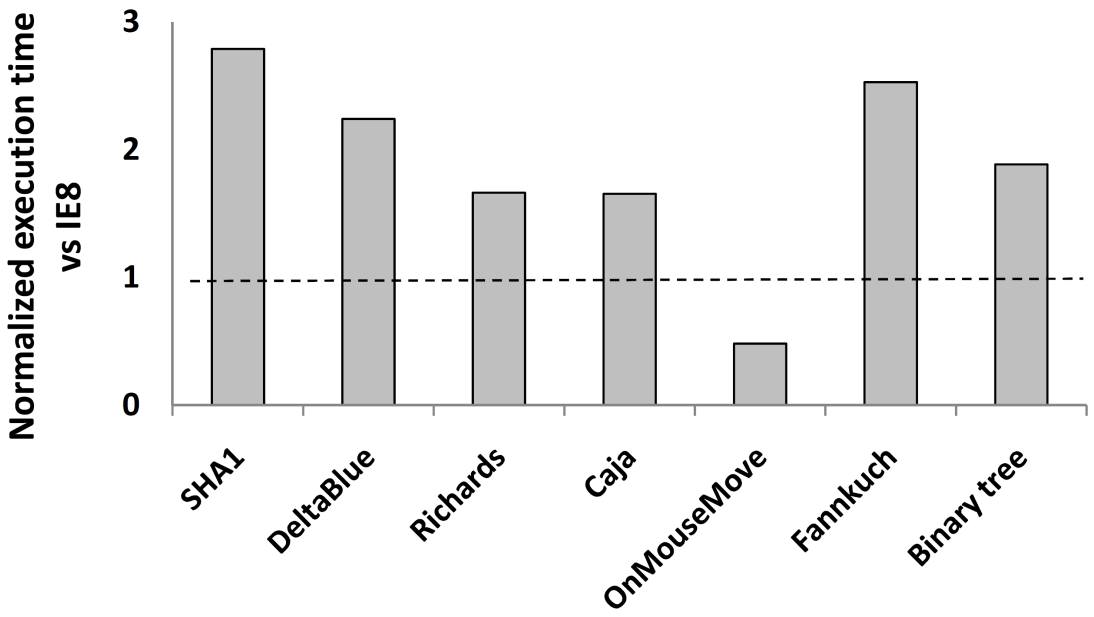


Figure 6.5: **Comparison of slowdown for Atlantis versus Internet Explorer 8 for several popular benchmarks.** All tests were CPU-bound except for `OnMouseMove`.

In summary, our prototype implementation of Atlantis is functional enough to load many Web pages and dispatch events at a fast rate. As mentioned in Section 6.6, we expect Atlantis's performance to improve significantly when we transition the code base from the default `.NET` runtime to the SPUR [38] runtime that is tuned for performance.

6.9 Related Work

Google Chrome [35], Gazelle [162], OP and OP2 [83, 84] are recent browser architectures that share the same security design principles, albeit with minor distinguishing features. For example, unlike OP2, Google Chrome does not enforce cross-frame protection mechanisms and security policy for plugin content. In Google Chrome, the rendering engine controls all network requests. This places critical security decisions in the same process as the rendering engine. OP2 separates the security enforcement from rendering and all policy enforcement is done within the browser kernel. Gazelle's architecture offers the same levels of protection as OP, except that it also focuses on providing cross-principal protection and fair sharing of all

system resources for all principals.

The closest piece of related work to Atlantis is Gazelle [162], which like Atlantis, is agnostic to the high-level runtime used by Web pages. Like Atlantis, Gazelle isolates all principal instances within separate isolation containers. However, Gazelle provides no protection *within* a container. In contrast, Atlantis uses C# `AppDomains` to isolate an instance's markup parser, layout engine, and Syphon interpreter. This restricts components to communication through message passing, improving robustness. It also allows individual components to be microrebooted on failure, as opposed to reloading the entire instance container.

Unlike Gazelle and OP, which rely on a single browser kernel, Atlantis uses `AppDomains` to place a unique kernel image in the address space of each principal instance. Each kernel can be sandboxed in ways that limit the damage a subverted kernel can inflict. In contrast, if Gazelle's single kernel is subverted, the attacker has complete access to the underlying machine.

ServiceOS [163] is an extension of Gazelle that implements new policies for resource allocation. Architecturally, ServiceOS is very similar to Gazelle, and differs from Atlantis in the same ways as Gazelle. Similarly, IBOS [155] extends the OP [83, 84] microkernel browsers, but unlike Atlantis, it relies on the same commodity black-box and grey-box software modules that currently affect the security and robustness of Web applications.

Cocktail [173] is another browser-based system for enhancing the security and reliability of Web browsers. However, unlike earlier systems, it takes a completely different approach and uses parallel execution of commodity Web browsers to defend against exploits targeting browser quirks and improve reliability against browser crashes.

Lastly, there are several JavaScript implementations of browser components like HTML parsers and JavaScript parsers [68, 78, 122, 143]. These libraries are typically used by a Web page to analyze markup or script source before it is passed to the browser's actual parsing engine or JavaScript runtime. Using extensible Web stacks, Atlantis lets pages extend and introspect the *real* application runtime. Atlantis' Syphon interpreter also provides new language primitives for making this introspection robust and efficient.

6.10 Summary

In this chapter, we have described Atlantis, a new Web browser that leverages exokernel principles not just for security, but for extensibility as well. While prior microkernel browsers reuse buggy and black-box components from monolithic browsers, Atlantis allows each Web page to define its own execution environment, like the markup parser, layout engine, DOM tree, and scripting runtime. Atlantis enables Web pages the freedom to tailor their execution environments without fear of breaking fragile browser interfaces. Our evaluation demonstrates this extensibility, and shows that our Atlantis prototype is fast enough to render popular pages and rapidly dispatch event handlers. Atlantis also leverages multiple kernels to provide stronger security guarantees than previous microkernel browsers.

Chapter 7

Conclusion

In this dissertation, we have shown that redesigning the Web platform leveraging operating system principles provides fine-grained security along with tremendous extensibility for the Web platform. Our work demonstrates that modern Web browsers can be retrofitted to provide security, however achieving true extensibility requires rearchitechting the Web browser. The work described in this dissertation is just a step towards development of the next-generation of the Web platform. We now describe future directions for further enhancements in security and extensibility of the Web platform.

7.1 Future Directions

Next-generation Web platform: The Web platform is currently huge, with users accessing the Internet using browsers from their desktops, laptops and mobile devices, However, the Web platform is due for another huge leap. In the near future, the Web ecosystem will include Internet-enabled augmented reality devices or heads-up displays (AR/HuD) [82, 116], consumer devices and a plethora of other mobile devices.

Internet-enabled consumer devices open up the possibilities of embedding browsers in user devices to facilitate rich interaction, like authentication with the devices over secure HTTP and personalization using the Web interface, or collaborative network troubleshooting tools for homes and enterprises. In contrast, Internet-enabled AR/HuD devices will combine a user's digital and real worlds to offer a unique experience. Development of rich applications for such platforms requires the use of novel browser abstractions that facilitate this process.

In the coming years, mobile devices are going to significantly outnumber desktops and laptops [45]. While the mobile hardware technology is moving rapidly, mobile Web browsers

are still in a nascent stage and offer a primitive browsing experience. Although orthogonal to the issue, application development for mobile platform is tedious and requires developers to program in different technologies for different target platforms. While a browser-based mobile platform can significantly ease the problem of application development on mobile platforms by allowing developers to program in standard Web technologies, it will take a radical redesign of the Web browsers to take advantage of the rapid changes in the mobile hardware technology.

Atlantis's exokernel approach provides an excellent starting point to develop solutions for all the above mentioned problems.

Security: In legacy browsers, exposing new and rich browser APIs along with new security primitives is usually a long drawn process since it involves reasoning about innumerable interactions with different browser subsystems. Moreover, adding new security primitives could itself be a cause of more browser vulnerabilities. A future direction could be to address the security implications of exporting rich functionality to Web applications by leveraging existing security primitives from different disciplines and introducing a comprehensive layered defense mechanism for containment or misuse of rich browser APIs. This approach would enable identification of a minimal set of such security primitives to secure all browser APIs, thereby allowing Web applications to access the rich APIs. A key challenge in developing such a system would be to define the role of the end user and development of appropriate interfaces to facilitate minimal user involvement in the security mechanism.

Techniques developed in Transcript have several applications beyond those we have explored, and these can be enhanced to be applicable for all browser APIs. For example, using appropriate security policies Transcript can be used to safe-guard access to all the rich browser APIs that are generally unavailable to Web applications today. This would involve understanding each browser API from a security standpoint and design relevant security policies to be implemented by Transcript.

User privacy: There has been very little effort to address the core problem of user privacy, i.e., to understand and characterize the ramifications of browser APIs on end-user privacy. Analysis of the privacy implications of various browser APIs could help to determine which compositions of APIs could leak end-user information to third parties, and engender development of

privacy-aware applications, say privacy-preserving advertisements. Such a study would also be useful in designing future browser APIs that are privacy-preserving by design. With a better understanding of privacy for browser APIs, a possible future direction would be to develop an in-browser framework that considers user expectations and interactions with applications to adapt the privacy levels in the Web browser.

7.2 Final Thoughts

Web development is undergoing a major transition from being exclusively Web-centric towards becoming more user-centric. However, to a large extent, today's Web browsers, which are poised to facilitate this enriched user experience, are far from being ready for this transition. This dissertation aims to contribute to the design and development of the next generation of Web browsers, and we have approached this goal by addressing the issues of security and extensibility in the modern Web platform.

Appendix A

Examples of Security Policies Implemented Using Transcript

Meyerovich and Livshits recently conducted a survey of the research literature to gather a list of security and reliability policies for untrusted JavaScript code [111]. Their list has seventeen wide-ranging policies that were enforced using the Conscript prototype. We were able to use Transcript to enforce sixteen of these policies. Figures A.1–A.14 show how fourteen of these policies will be enforced using iblocks in a Transcript-enhanced Web application.

Two other policies, which prevent the inclusion of dynamic scripts and inline scripts (#1 and #3 in [111]), are enforced by design in Transcript. Dynamic scripts refer to those scripts included using a `<script>` element, while inline scripts refer to those that can be included via `document.write` without a `src` attribute. Transcript prevents these cases by construction because it requires third party code to be included using a modified `script` tag (Section 5.8.4). It prevents inline scripts because they must be recognized by the HTML parser and glued within the iblock. Transcript can prevent these scripts by simply not gluing this code when the transaction resumes. We do not illustrate one of their policies, (#9 "Whitelist cross-domain requests" in [111]), because it uses `XDomainRequest`, which is specific to Internet Explorer. However, Transcript was able to enforce a closely-related policy for Firefox, which used `XMLHttpRequest` instead.

```
1 var arg = tx.getArgs();
2 if((tx.getCause().match("setTimeout") ||
3   tx.getCause().match("setInterval")) &&
4   arg[0] instanceof Function) {
5   // perform action on behalf of untrusted code
6 }
```

Figure A.1: **No string arguments to `setInterval`, `setTimeout`.** This policy checks the cause of a transaction suspend to be either `setInterval` or `setTimeout` and verifies that the first argument passed to them is an instance of the Function object.

```
1 var cause = tx.getCause();
2 var arg = tx.getArgs();
3 if(cause.match("appendChild") &&
4  arg[0].nodeName.match("SCRIPT")) {
5   var src = arg[0].getAttribute("src");
6   if(isWhiteListed(src)) {
7     // perform action on behalf of untrusted code
8   }
9 }
```

Figure A.2: **Script tag whitelist.** This policy checks if the `src` attribute of a dynamically inserted `<script>` tag is permitted by a white-list. `<script>` tags passed as arguments to `document.write` can be checked for white-listed `src` URLs before their "gluing" within the introspection block. The helper method `isWhiteListed` returns `true` if its argument represents a white-listed URL.

```
1 var arg = tx.getArgs();
2 if(tx.getCause().match("appendChild") &&
3  arg[0].nodeName.match("SCRIPT" ) &&
4  !parentNodeHasNoScript(arg[0])) {
5   // perform action on behalf of untrusted code
6 }
```

Figure A.3: **NO SCRIPT tag.** This policy verifies that a `<script>` tag is inserted dynamically only if it is not within a `<nodynamicscript>` tag. Execution of inline `<script>`s introduced by `document.write` can be prevented within the introspection block by not "gluing" them. The helper method `parentNodeHasNoScript` returns `true` if the its argument has a `<nodynamicscript>` tag as an ancestor.

```
1 if((tx.getObject() instanceof XMLHttpRequest) &&
2   tx.getCause().match("open") &&
3   (tx.getArgs()[3] || tx.getArgs()[4]) &&
4   tx.getArgs()[1].substr(0,8).match("https://")) {
5   // perform action on behalf of untrusted code
6 }
```

Figure A.4: **Restrict `XMLHttpRequest` to secure connections.** This policy ensures that if a username or password is supplied, then the second argument passed to the `open` method of the `XMLHttpRequest` object must represent a secure URL.

```
1 var rs = tx.getReadSet();
2 var ws = tx.getWriteSet();
3 if(rs.checkMembership(document,"cookies") ||
4   ws.checkMembership(document,"cookies")) {
5   // do not commit the transaction
6 }
```

Figure A.5: **HTTP-only cookies.** This policy prevents untrusted JavaScript from making persistent changes if it has accessed domain cookies.

```
1 if((tx.getObject() instanceof Window) &&
2   tx.getCause().match("postMessage") &&
3   isWhitelisted(tx.getArgs()[1])) {
4   // perform action on behalf of untrusted code
5 }
```

Figure A.6: **Whitelist cross-frame messages.** This policy inspects the destination URL for cross-frame messages and permits message communication only if the URL is white-listed. The helper method `isWhiteListed` returns `true` if its argument represents a white-listed URL.

```
1 var rs = tx.getReadSet();
2 var arg = tx.getArgs();
3 var ok = !rs.checkMembership(document,"cookies");
4 if(tx.getCause().match("setAttribute") &&
5   tx.getObject().nodeName.match("A") &&
6   arg[0].match("href") &&
7   (ok || !checkForeignDomain(arg[1]))) {
8   // perform action on behalf of untrusted code
9 }
```

Figure A.7: **No foreign links after a cookie access.** This policy prevents dynamically setting the `href` attribute of the `<A>` tag to a foreign domain if the domain cookies have been accessed. The helper method `checkForeignDomain` returns `true` if its argument represents a foreign domain.

```
1 if((tx.getObject() instanceof Window) &&
2   tx.getCause().match("open")) {
3     if((count++ < 2) &&
4       hasCompliantDimensions(tx.getArgs()[2]))
5       // perform action on behalf of untrusted code
6 }
```

Figure A.8: **Limit popup window creation.** This policy limits the number of popup windows that can be opened. The policy also ensures that the popup window shows up only if it has compliant dimensions. The helper method `hasCompliantDimensions` returns `true` if the window dimensions are allowed.

```
1 var arg = tx.getArgs();
2 if(tx.getCause().match("createElement") &&
3   !args[0].match("IFRAME")) {
4   // perform action on behalf of untrusted code
5 }
```

Figure A.9: **Disable dynamic `IFRAME` creation.** This policy disables dynamic `<iframe>` creation by not completing the `createElement` request within the introspection block if the argument type matches `<IFRAME>`.

```
1 var ws = tx.getWriteSet();
2 if(ws.checkMembership(document,"location")){
3   var loc = ws.getValue(document,"location");
4   if(!isWhiteListed(loc))
5     // do not commit the transaction
6 }
```

Figure A.10: **Whitelist URL redirection.** This policy inspects the redirection URL and commits the transaction only if the URL is white-listed. The helper method `isWhiteListed` returns `true` if its argument represents a white-listed URL.

```
1 var cause = tx.getCause();
2 if(!cause.match("alert") &&
3   !cause.match("prompt")) {
4   // perform action on behalf of untrusted code
5 }
```

Figure A.11: **Prevent resource abuse.** This policy disables all `alert` or `prompt` notifications from the untrusted code.

```
1 var arg = tx.getArgs();
2 if (tx.getCause().match("$")) {
3   if (tx.getSource().match("jQuery.js")) {
4     var regex = /[a-zA-Z0-9.#:]+(( > |)[a-zA-Z0-9.#:]+)+$/;
5     if(arg[0].match(regex))
6       // perform action on behalf of untrusted code
7   }
8 }
```

Figure A.12: **Simple and fast jQuery selectors.** This policy allows only selectors with fast composition operators.

```
1 var arg = tx.getArgs();
2 if (tx.getCause().match("$")) {
3   if (tx.getSource().match("jQuery.js")) {
4     var nodes = $(arg[0], arg[1]);
5     if (!nodes.length)
6       // do not commit transaction
7   }
8 }
```

Figure A.13: **Explicit jQuery selector failure.** This policy allows a library user to attach behavior to the when the selector fails.

```
1 // init_complete initialized to false
2 if (tx.getCause().match("eval")){
3   if (tx.getSource().match("jQuery.js") &&
4     !init_complete) {
5       init_complete = true;
6       // perform action for untrusted code
7     }else
8       JSON.parse(tx.getArgs()[0]);
9 }
```

Figure A.14: **Staged `eval` restrictions.** The policy allows "jQuery.js" to initialize itself using `eval` but uses `JSON.parse` for every other `eval` request.

Bibliography

[1] Addthis. http://www.addthis.com/.

[2] BIGACE web content management system. http://www.bigace.de/.

[3] Cross-Site Scripting Worm Hits MySpace. http://betanews.com/2005/10/13/cross-site-scripting-worm-hits-myspace/.

[4] Customizable shortcuts. https://addons.mozilla.org/en-US/firefox/addon/customizable-shortcuts/L.

[5] Cve-2012-5836. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5836.

[6] Document object model. http://www.w3.org/DOM.

[7] Dom-based xss injection. https://www.owasp.org/index.php/Interpreter_Injection#DOM-based_XSS_Injection.

[8] Harmony Proxies. http://wiki.ecmascript.org/doku.php?id=harmony:proxies.

[9] Information disclosure (mouse tracking) vulnerability in microsoft internet explorer versions 6-10. http://seclists.org/bugtraq/2012/Dec/81.

[10] Internet Explorer 8. http://www.microsoft.com/windows/internet-explorer.

[11] Jetpack SDK. https://addons.mozilla.org/en-US/developers/docs/sdk/latest/.

[12] jQuery: The write less, do more, JavaScript library. http://jquery.com.

[13] Jquery UI slider plugin. http://jqueryui.com/demos/slider.

[14] JavaScript widgets/menu. http://jswidgets.sourceforge.net.

[15] Microsoft web sandbox. http://websandbox.livelabs.com/.

[16] Mozilla Jetpack. https://wiki.mozilla.org/Jetpack.

[17] Netscape Plugin Application Programming Interface. http://en.wikipedia.org/wiki/NPAPI.

[18] Principle of Least Authority. http://en.wikipedia.org/wiki/Principle_of_least_privilege.

[19] Signed scripts in Mozilla: JavaScript privileges. http://www.mozilla.org/projects/security/components/signed-scripts.html.

[20] SunSpider JavaScript Benchmark. http://www.webkit.org/perf/sunspider/sunspider.html.

[21] Twitter/profile widget. http://twitter.com/about/resources/widgets/widget_profile.

[22] XML user interface language (XUL) project. http://www.mozilla.org/projects/xul.

[23] FormSpy: McAfee avert labs, July 2006. http://vil.nai.com/vil/content/v_140256.htm.

[24] Mozilla Firefox Firebug extension—Cross-zone scripting vulnerability, April 2007. http://www.xssed.org/advisory/33.

[25] FFsniFF: FireFox sniFFer, June 2008. http://azurit.elbiahosting.sk/ffsniff.

[26] Firefox add-ons infecting users with trojans, May 2008. http://www.webmasterworld.com/firefox_browser/3644576.htm.

[27] Trojan.PWS.ChromeInject.B, Nov 2008. http://www.bitdefender.com/VIRUS-1000451-en--Trojan.PWS.ChromeInject.B.html.

[28] ECMAScript language spec., ECMA-262, 5^{th} edition, Dec 2009.

[29] Netscape Navigator 3.0. Using data tainting for security. http://www.aisystech.com/resources/advtopic.htm.

[30] Devdatta Akhawe, Prateek Saxena, and Dawn Song. Privilege separation in html5 applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 23–23, Berkeley, CA, USA, 2012. USENIX Association.

[31] J. Albahari and B. Albahari. *C# 3.0 in a Nutshell*. O'Reilly Publishing, O'Reilly Media, Inc., 3rd edition, 2007.

[32] Fabio Assolini. Think twice before installing chrome extensions. http://www.securelist.com/en/blog/208193414/Think_twice_before_installing_Chrome_extensions.

[33] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 22–22, Berkeley, CA, USA, 2010. USENIX Association.

[34] Sruthi Bandhakavi, Nandit Tiku, Wyatt Pittman, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 54(9):91–99, September 2011.

[35] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *In Proceedings of the Network and Distributed System Security Symposium*, 2010.

[36] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 305–314, New York, NY, USA, 2005. ACM.

[37] Philippe Beaucamps and Daniel Reynaud. Malicious Firefox Extensions. In *Symposium sur la sécurité des techniques d'information et de communication*, Rennes, France, 2008.

[38] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: a trace-based jit compiler for cil. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010. ACM.

[39] Arnar Birgisson, Mohan Dhawan, Úlfar Erlingsson, Vinod Ganapathy, and Liviu Iftode. Enforcing authorization policies using transactional memory introspection. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 223–234, New York, NY, USA, 2008. ACM.

[40] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, and Yan Chen. Virtual browser: a web-level sandbox to secure third-party javascript without sacrificing functionality (poster). In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 654–656, New York, NY, USA, 2010. ACM.

[41] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[42] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 143–163, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[44] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.

[45] Cisco. Number of mobile devices to hit 8 billion by 2016. http://news.cnet.com/8301-13506_3-57377325-17/number-of-mobile-devices-to-hit-8-billion-by-2016-cisco-says/.

[46] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.

[47] Chris Coyier. Percentage Bugs in WebKit. *CSS-tricks Blog*. http://css-tricks.com/percentage-bugs-in-webkit/, August 30, 2010.

[48] Steven Crites, Francis Hsu, and Hao Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 99–108, New York, NY, USA, 2008. ACM.

[49] D. Crockford. ADsafe - Making JavaScript safe for advertising. http://adsafe.org.

[50] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, July 2006.

[51] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript. In *Proceedings of the 2nd conference on USENIX Workshop on offensive technologies*, WOOT'08, pages 1:1–1:6, Berkeley, CA, USA, 2008. USENIX Association.

[52] Dasient. Structural Vulnerabilities on Websites: Why Enterprise Websites Are Vulnerable to Malware Attacks. http://info.dasient.com/structural_vulnerabilities.html.

[53] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 748–759, New York, NY, USA, 2012. ACM.

[54] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 535–544, New York, NY, USA, 2008. ACM.

[55] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[56] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC '09, Washington, DC, USA, 2009. IEEE Computer Society.

[57] Mohan Dhawan, Chung-chieh Shan, and Vinod Ganapathy. The case for javascript transactions: position paper. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 6:1–6:7, New York, NY, USA, 2010. ACM.

[58] Mohan Dhawan, Chung-chieh Shan, and Vinod Ganapathy. Enhancing javascript with transactions. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 383–408, Berlin, Heidelberg, 2012. Springer-Verlag.

[59] Vladan Djeric and Ashvin Goel. Securing script-based extensibility in web browsers. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 23–23, Berkeley, CA, USA, 2010. USENIX Association.

[60] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 297–306, New York, NY, USA, 2011. ACM.

[61] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 339–354, Berkeley, CA, USA, 2008. USENIX Association.

[62] Ecma International. Ecmascript language specification, 5^{th} edition, December 2009.

[63] H. Edskes. IE8 overflow and expanding box bugs. *Final Builds Blog*.http://www.edskes.net/ie/ie8overflowandexpandingboxbugs.htm, 2010.

[64] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.

[65] B. Eich. Better security for JavaScript, March 2009. Dagstuhl Seminar 09141: Web Application Security.

[66] B. Eich. JavaScript security: Let's fix it, May 2009. Web 2.0 Security and Privacy Workshop.

[67] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[68] Envjs Team. Envjs: Bringing the Browser. http://www.envjs.com/, 2010.

[69] Úlfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Ithaca, NY, USA, 2004. AAI3114521.

[70] eSpace Technologies. A tiny bug in Prototype JS leads to major incompatibility with Facebook JS client library. *eSpace.com blog*, April 23, 2008.

[71] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[72] Facebook. FBJS - Facebook developerwiki. 2007.

[73] M. Felleisen. *The Calculi of λ_v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, 1987.

[74] Fielding, R., Gettys, J., Mogul, J.,Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[75] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure JavaScript subsets. In *In Proceedings of the Network and Distributed System Security Symposium*, 2010.

[76] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 5th edition, 2006.

[77] Forrester Consulting. *eCommerce Web Site Performance Today: An Updated Look At Consumer Reaction To A Poor Online Shopping Experience*. White paper, 2009.

[78] Daniel Glazman. JSCSSP: A CSS parser in JavaScript. http://www.glazman.org/JSCSSP/, 2010.

[79] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. pages 213–223, 2005.

[80] Google. Chrome Web Store. https://chrome.google.com/webstore/category/extensions.

[81] Google. V8 Benchmark Suite (version 5). http://v8.googlecode.com/svn/data/benchmarks/v5/run.html, 2010.

[82] Goolge. Project Glass. https://plus.google.com/+projectglass/posts.

[83] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.

[84] Chris Grier, Shuo Tang, and Samuel T. King. Designing and implementing the op and op2 web browsers. *TWEB*, 5(2):11, 2011.

[85] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[86] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187, New York, NY, USA, 2011. ACM.

[87] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 115–130, Washington, DC, USA, 2011. IEEE Computer Society.

[88] 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.

[89] Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '05, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[90] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin css attacks. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 619–629, New York, NY, USA, 2010. ACM.

[91] Lon Ingram and Michael Walfish. Treehouse: Javascript sandboxes to helpweb developers help themselves. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 13–13, Berkeley, CA, USA, 2012. USENIX Association.

[92] J. Zaytsev. What's wrong with extending the DOM. *Perfection Kills Website*. http://perfectionkills.com/whats-wrong-with-extending-the-dom, April 5, 2010.

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

[94] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 601–610, New York, NY, USA, 2007. ACM.

[95] jQuery Message Forum. Focus() inside a blur() handler. https://forum.jquery.com/topic/focus-inside-a-blur-handler, January 2010.

[96] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-chieh Shan. An analysis of the mozilla jetpack extension framework. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 333–355, Berlin, Heidelberg, 2012. Springer-Verlag.

[97] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[98] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 26–37, New York, NY, USA, 2006. ACM.

[99] L. Lazaris. CSS Bugs and Inconsistencies in Firefox 3.x. *Webdesigner Depot*. http://www.webdesignerdepot.com/2010/03/css-bugs-and-inconsistencies-in-firefox-3-x, March 15, 2010.

[100] Guanhua Yan Lei Liu, Xinwen Zhang and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *In Proceedings of the Network and Distributed System Security Symposium*, 2012.

[101] Zhuowei Li, XiaoFeng Wang, and Jong Youl Choi. Spyshield: preserving privacy from spy add-ons. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID'07, pages 296–316, Berlin, Heidelberg, 2007. Springer-Verlag.

[102] Kaisen Lin, David Chu, James Mickens, Li Zhuang, Feng Zhao, and Jian Qiu. Gibraltar: exposing hardware devices to web pages using ajax. In *Proceedings of the 3rd USENIX conference on Web Application Development*, WebApps'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[103] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. Adjail: practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association.

[104] Mike Ter Louw, Jin Soon Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.

[105] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society.

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

[107] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 505–522, Berlin, Heidelberg, 2009. Springer-Verlag.

[108] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 125–140, Washington, DC, USA, 2010. IEEE Computer Society.

[109] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 77–91, Washington, DC, USA, 2009. IEEE Computer Society.

[110] Leo A. Meyerovich, Adrienne Porter Felt, and Mark S. Miller. Object views: fine-grained sharing in browsers. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 721–730, New York, NY, USA, 2010. ACM.

[111] Leo A. Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 481–496, Washington, DC, USA, 2010. IEEE Computer Society.

[112] James Mickens and Mohan Dhawan. Atlantis: robust, extensible execution environments for web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 217–231, New York, NY, USA, 2011. ACM.

[113] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX conference on*

Networked systems design and implementation, NSDI'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[114] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. Crom: Faster web browsing using speculative execution. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[115] James Mickens and Matthew Finifter. Jigsaw: efficient, low-effort mashup isolation. In *Proceedings of the 3rd USENIX conference on Web Application Development*, WebApps'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[116] Microsoft. Event augmentation with real-time information. http://appft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PG01&p=1&u=2Fnetahtml%2FPTO%2Fsrchnum.html&r=1&f=G&l=50&s1=%2220120293548%22.PGNR.&OS=DN/20120293548&RS=DN/20120293548.

[117] Microsoft. Update for Native JSON feature in IE8. http://support.microsoft.com/kb/976662, February 2010.

[118] Microsoft Developer Network. DOM Level 2 - Events. http://msdn.microsoft.com/en-us/library/ff943604%28v=vs.85%29.aspx.

[119] Mark Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. Draft specification, January 15, 2008.

[120] Mozilla. Mozilla Addon Statistics Dashboard. https://addons.mozilla.org/en-US/statistics/addons_in_use/?last=30.

[121] Mozilla Addon SDK. Content Proxy. https://addons.mozilla.org/en-US/developers/docs/sdk/latest/packages/api-utils/content/proxy.html.

[122] Mozilla Corporation. Narcissus javascript. http://mxr.mozilla.org/mozilla/source/js/narcissus/.

[123] Mozilla Developer Center. HTTP access control. http://developer.mozilla.org/En/HTTP_access_control.

[124] Mozilla Developer Network. Electrolysis/Firefox. https://wiki.mozilla.org/Electrolysis/Firefox.

[125] Mozilla Developer Network. XPCNativeWrapper. https://developer.mozilla.org/en/docs/XPCNativeWrapper.

[126] National Vulnerability Database. CVE-2010-2301, 2010. Cross-site scripting vulnerability: innerHTML.

[127] New York Times. Note to Readers. http://www.nytimes.com/2009/09/13/business/media/13note.html?_r=0.

[128] T. Olsson. *The Ultimate CSS Reference*. Sitepoint, Collingwood, Victoria, Austraiia, 2008.

[129] Orangoo-Labs. GoogieSpell. http://orangoo.com/labs/GoogieSpell.

[130] Orangoo-Labs. GreyBox. http://orangoo.com/labs/GreyBox.

[131] Orangoo-Labs. Sortable list widget. http://orangoo.com/AJS/examples/sortable_list.html.

[132] S. Di Paola and G. Fedon. Subverting Ajax: Next generation vulnerabilities in 2.0 Web applications. In *23rd Chaos Communication Congress*, 2006.

[133] Terrence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, Raleigh, North Carolina, 2007.

[134] Peter-Paul Koch. QuirksMode–for all your browser quirks. http://www.quirksmode.org, 2011.

[135] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[136] M. Pilgrim. Greasemonkey for secure data over insecure networks/sites, July 2005. http://mozdev.org/pipermail/greasemonkey/2005-July/003994.html.

[137] J. Pobar, T. Neward, D. Stutz, and G. Shilling. Shared Source CLI 2.0 Internals. http://callvirt.net/blog/files/Shared%20Source%20CLI%202.0%20Internals.pdf, 2008.

[138] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[139] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 161–176, New York, NY, USA, 2009. ACM.

[140] Emil Protalinski. Malicious chrome extensions hijack facebook accounts. http://www.zdnet.com/blog/security/malicious-chrome-extensions-hijack-facebook-accounts/11074.

[141] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[142] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3), September 2007.

[143] John Resig. Pure JavaScript HTML Parser. http://ejohn.org/blog/pure-javascript-html-parser/, May 2008.

[144] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[145] J. Ruderman. The same-origin policy, August 2001. http://www.mozilla.org/projects/security/components/same-origin.html.

[146] Secunia Advisory SA24743/CVE-2007-1878/CVE-2007-1947. Mozilla Firefox Firebug extension two cross-context scripting vulnerabilities.

[147] Secunia Advisory SA30284. FireFTP extension for Firefox directory traversal vulnerability.

[148] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 213–227, New York, NY, USA, 1996. ACM.

[149] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[150] 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.

[151] Christian Stork, Peter Housel, Vivek Haldar, Niall Dalton, and Michael Franz. Towards language-agnostic mobile code. In *Proceedings of the Workshop on Multi-Language Infrastructure and Interoperability*, Firenze, Italy, 2001.

[152] Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *In Proceedings of the Network and Distributed System Security Symposium*, pages 265–278, 2005.

[153] Symantec. Vulnerability Trends. http://www.symantec.com/threatreport/topic.jsp?id=vulnerability_trends&aid=web_browser_plug_in_vulnerabilities.

[154] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 363–378, Washington, DC, USA, 2011. IEEE Computer Society.

[155] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the illinois browser operating system. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[156] Chris Tyler. *X Power Tools*. O'Reilly Media, Inc., Cambridge, MA, 2007.

[157] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS07*, 2007.

[158] W3C. Same Origin Policy. http://www.w3.org/Security/wiki/Same_Origin_Policy.

[159] W3C Web Apps Working Group. Web Storage: W3C Working Draft. http://www.w3.org/TR/2009/WD-webstorage-20091029, October 29, 2009.

[160] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[161] 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.

[162] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 417–432, Berkeley, CA, USA, 2009. USENIX Association.

[163] Helen J. Wang, Alexander Moshchuk, and Alan Bush. Convergence of desktop and web applications on a multi-service os. In *Proceedings of the 4th USENIX conference on Hot topics in security*, HotSec'09, pages 11–11, Berkeley, CA, USA, 2009. USENIX Association.

[164] Hironori Washizaki, Atsuto Kubo, Tomohiko Mizumachi, Kazuki Eguchi, Yoshiaki Fukazawa, Nobukazu Yoshioka, Hideyuki Kanuka, Toshihiro Kodaka, Nobuhide Sugimoto, Yoichi Nagai, and Rieko Yamamoto. Aojs: aspect-oriented javascript programming framework for web development. In *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '09, pages 31–36, New York, NY, USA, 2009. ACM.

[165] Web Hypertext Application Technology Working Group (WHATWG). Web Workers (Draft Recommendation). http://www.whatwg.org/specs/web-workers/current-work/, September 10, 2010.

[166] S. Willison. Understanding the Greasemonkey vulnerability, July 2005. http://simonwillison.net/2005/Jul/20/vulnerability.

[167] World Wide Web Consortium. Document object model (DOM) level 2 core specification. W3C Recommendation, November 13, 2000.

[168] World Wide Web Consortium. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. W3C Working Draft. http://www.w3.org/TR/CSS2, September 8, 2009.

[169] World Wide Web Consortium. Geolocation API Specification. http://dev.w3.org/geo/api/spec-source.html, February 10, 2010.

[170] World Wide Web Consortium. HTML Device: An addition to HTML. http://dev.w3.org/html5/html-device/, September 9, 2010.

[171] World Wide Web Consortium. HTML5: A vocabulary and associated APIs for HTML and XHTML. W3C Working Draft. http://www.w3.org/TR/html5, June 24, 2010.

[172] WWW-Consortium. Document object model events, Nov 2000. http://www.w3.org/TR/DOM-Level-2-Events/events.html.

[173] Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using replicated execution for a more secure and reliableweb browser. In *In Proceedings of the Network and Distributed System Security Symposium*, 2012.

[174] Yahoo! Yahoo! User Interface Library. http://yuilibrary.com/.

[175] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[176] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 237–249, New York, NY, USA, 2007. ACM.

[177] Saman Zarandioon, Danfeng (Daphne) Yao, and Vinod Ganapathy. Omos: A framework for secure communication in mashup applications. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 355–364, Washington, DC, USA, 2008. IEEE Computer Society.