

TOWARDS REDESIGNING WEB BROWSERS WITH SECURITY PRINCIPLES

DONG XINSHU
(B.Eng, East China Normal University)

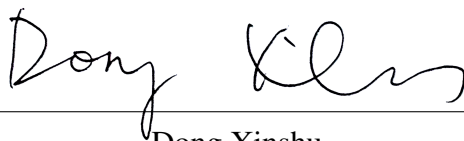
**A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE**

2013

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Dong Xinshu

18 Jan 2013

ACKNOWLEDGEMENTS

I am very grateful to have the opportunity to take on my PhD journey in National University of Singapore. It is long and tough, but I am really blessed. Without His blessings, I would not have benefited so much from these years with great professors and colleagues.

Prof. Zhenkai Liang is never just my *thesis advisor*. He enlightens and encourages me far beyond technical aspects. His insight and experience in computer systems helps me a lot in my research projects, but more importantly, I learn from him the perseverance and determination in following research interest and striving for excellence. It took me quite some time and effort to transform from an average graduate to a security researcher, and Prof. Liang has always been inspiring and patient in guiding me through.

I also want to express my special thanks to my buddy during my PhD life, Kailas Patil. Tough research problems become easier when we collaborate, and tough days become shorter when we work and play together.

Several professors in and out of the department have provided me insightful guidance and direction. I would like to thank all of them: Prof. Roland Yap, Prof. Prateek Saxena, Prof. Jian Mao, Prof. Xuxian Jiang, Prof. Ee-chien Chang, and Prof. Hon Wai Leong, as well as my collaborators Dr. Xuhui Liu and Minh Tran.

My thanks also extend to my lab-mates: Utsav Saraf, Jiengang Wang, Mingwei Zhang, Chengfang Fang, Xiaolei Li, Ting Dai, Chunwang Zhang, Hong Hu, Bodhisatta Barman Roy, Zhaofeng Chen, Hossein Siadati, Shruti Tople, Shweta Shinde, and many others. With you guys, our lab never lacks cheerful and exciting moments.

I also have vivid memory in my earlier days, how Mr. Jing Zhou, one of the few experts then in my hometown, voluntarily taught me the most basic things about computers during several summers. My father, besides all his love for me, invested a great deal to buy me computers and hardware just to encourage me to pursue my interest, when they were still luxuries. I can never repay the love and care from my parents.

Finally I am in full gratitude to my beloved wife, Ying Li, who has been persistently supportive when I encounter hardness in my research, and generously forgiving when I spend days and evenings in the lab.

Contents

1	Introduction	1
2	Background on Browser Security: Threats & Defenses	6
2.1	The Web Browser	6
2.2	Threats to Web Clients & Existing Countermeasures	9
2.2.1	Threats to Web Applications & Existing Defenses	9
2.2.2	Threats to Web Browsers & Existing Defenses	12
2.3	New Security Requirements from Modern Web Applications	12
3	ADSENTRY: Comprehensive and Flexible Confinement of JavaScript-based Advertisements	14
3.1	Problem Overview	15
3.2	System Design	16
3.2.1	Shadow JavaScript Engine	18
3.2.2	Page Agent	18
3.2.3	Policy Enforcer	20
3.3	Implementation	20
3.3.1	Specifying Advertisement Scripts	21
3.3.2	Shadow JavaScript Runtime and Virtual DOM	21
3.3.3	Page Agent	22
3.3.4	Access Control Policy Enforcement	24
3.4	Evaluation	26
3.4.1	Browser Exploits	27
3.4.2	Script Injection by Ads	27
3.4.3	Privacy Protection	28
3.4.4	Performance Evaluation	30
3.5	Discussion	31
3.6	Related Work	31
3.7	Summary	33
4	A Quantitative Evaluation of Privilege Separation in Web Browser Designs	34
4.1	Overview	37
4.1.1	Privilege Separation in Concept	37
4.1.2	Privilege Separation in Browsers	38
4.2	Quantifying Trade-offs with Empirical Measurements	40

4.2.1	Security Parameters	40
4.2.2	Performance Parameters	41
4.3	Implementation of PRIVGAUGE	44
4.3.1	Challenge: Handling “Bridge” Components	45
4.4	Experimental Evaluation	46
4.4.1	Measurement Goals	46
4.4.2	Measurement over Alexa Top 100 Websites	46
4.4.3	Summary of Findings	52
4.4.4	Discussion	55
4.5	Related Work	56
4.6	Summary	56
5	BSM: Towards General Security Support in Browsers	58
5.1	Requirements for Flexible Security in Web Browsers	59
5.2	Design of Browser Security Modules	61
5.2.1	BSM Architecture	62
5.2.2	Fine-grained Principal Support and Tracking	62
5.2.3	Accurate Browser States and Context Information	63
5.3	Experiments and Evaluation	65
5.3.1	Implementation	65
5.3.2	Supporting Security Requirements	65
5.3.3	Case Studies: Supporting Existing Security Solutions	66
5.3.4	Performance	73
5.3.5	Supporting New Security Applications	74
5.4	Discussion and Future Work	76
5.5	Related Work	77
5.5.1	New Web Browser Architecture	77
5.5.2	Defenses to Attacks on Web Applications	77
5.5.3	Operating System Security	78
5.6	Summary	78
6	Protecting Sensitive Web Content from Client-side Vulnerabilities with	
	CRYPTONS	80
6.1	Problem & Overview	83
6.1.1	Threat Model	83
6.1.2	CRYPTON: A New Abstraction	84
6.1.3	Design Overview	86
6.1.4	Out-of-scope Threats	88
6.2	Design	89
6.2.1	CRYPTON Definition	89
6.2.2	CRYPTON-KERNEL Design	90
6.2.3	Security Invariants	93
6.2.4	Key Techniques and Security Analysis	94

6.2.5	Prototype & Deployment	96
6.2.6	Alternative Deployment into Google Chrome	97
6.3	Evaluation	98
6.3.1	Applicability to Real-world Applications	98
6.3.2	Reduction in Attack Surface	101
6.3.3	Performance	102
6.3.4	Study on Firefox Vulnerabilities	103
6.4	Related Work	104
6.4.1	Enhancement of Browser Security Mechanisms	104
6.4.2	Cryptographic Techniques	105
6.5	Summary	105
7	Conclusion	106

Summary

Modern web browsers serve as a run-time platform for executing web applications. New technologies built into web browsers have enabled a new generation of interactive and responsive web applications that are more powerful than ever. However, the lagging progress in enhancing browser security has made various attacks possible, such as cross-site scripting (XSS), cross-site request forgery (CSRF), clickjacking in web applications, as well as drive-by download, buffer overflow, and heap spray attacks against browsers.

Many existing approaches address specific security vulnerabilities or particular types of attacks; however, new features are constantly being introduced into web browsers, often giving rise to new security issues. Thus, reactive approaches are insufficient in this era of rapid evolution in browser functionality. Instead, we need more proactive and principled approaches to redesign web browsers as a more secure platform for online transactions and sensitive data. In this thesis, we systematically analyze the existing execution environment in web browsers. Our analysis finds that traditional security principles [142] that are protecting our operating systems and critical programs today have not been profoundly embodied in web browsers. Such principles include *separation of privilege*, *least privilege*, *complete mediation*, and the often overlooked *economy of mechanism*, etc. We thus develop a line of novel solutions to bring in these security principles into web browsers, effectively combating threats in modern web applications and browsers.

We propose a new JavaScript isolation primitive to transparently isolate untrusted JavaScript-based advertisements (ads). It allows web applications to securely embed such untrusted ads, without risking the integrity and confidentiality of their application logic and user data. As there usually exist various program partitioning alternatives in privileged-separated browser designing, we develop a measurement-driven methodology to quantify security-performance trade-offs in browser designs. This methodology can help browser designers identify potential performance bottlenecks in their designs that would require additional optimization. Even with privilege-separated web browser designs, security incidents could occur within a program partition. We thus propose a general security framework to support intra-partition application behavior monitoring and control. With more sensitive data being processed in online transactions, browsers need to provide strong protection to secure such data. We propose a new abstraction for data-centric isolation, which provides complete mediation on access to sensitive data, and eliminates most of client-side code in browsers out of the Trusted Computing Base (TCB).

Solutions proposed in this thesis demonstrate their effectiveness and practicality in real-world scenarios. We foresee that this thesis will evoke more fundamental rethink and redesign of web browsers with comprehensive security support to modern web applications.

List of Figures

2.1	Browser Internals	7
3.1	An Architecture Overview of ADSENTRY	17
3.2	A Firebug-based Tool to Facilitate Specifying Access Permissions for Ads	29
4.1	Overview of Privilege Separation	37
4.2	Gray-scaled Chart of Call Counts across Code Units. <i>Components are numbered as with Table 4.3.</i> Each cell at (i, j) corresponds to the call counts between Code Unit i and j	47
4.3	Occurrence Frequencies of Unique Pairs of Different Requestor-Destination Origins <i>746 unique pairs only occur once, while only 164 unique pairs occur more than 15 times.</i>	51
5.1	The Architecture of BSM	61
5.2	MashupOS: New Primitives	68
5.3	MashupOS: Specific Access Control Example	68
5.4	MashupOS: <OpenSandbox>	69
5.5	MashupOS: CommRequest in JavaScript	69
5.6	MashupOS: JavaScript Insertion	69
5.7	MashupOS: Primitive Translation	70
5.8	Noncespaces: Policy Enforcement and Prefix Parsing	72
5.9	A Sample Web Page Snippet	74
5.10	Graph Showing Request Dependency for The Sample Web Page Snippet	75
5.11	A Sample Behavior Model	76

6.1	Overview of the CRYPTON-KERNEL, <i>consisting of 4 components in gray, and the sandbox of a dashed red rectangle.</i>	90
6.2	CRYPTON Tags Embedded in HTML. <i>An example of search emails matching user-input keywords, including the CRYPTON header, functions, and information blocks.</i>	91
6.3	Sequence Diagram for CRYPTON-KERNEL Operations	92
6.4	TEMI: a Reduced Execution Environment for CRYPTON Functions to Access Sensitive Info	95
6.5	Number of User Input Fields and CRYPTON Functions Required in Signup Pages	101
6.6	Performance Overhead of CRYPTON-KERNEL with Dromaeo Benchmarks & Manual Test Page, <i>with varying portions of sensitive texts</i> . . .	102

List of Tables

3.1	ADSENTRY Evaluation Using Browser Exploits	26
3.2	ADSENTRY Evaluation Using JavaScript Injection Attacks	27
3.3	Websites Used in User Experience Evaluation	29
3.4	Runtime Page Load Overhead of ADSENTRY	30
4.1	Privilege Separation in Browsers <i>The table explains different partitioning dimensions in browser designs. For the right part of the table, same symbols denote the corresponding components are in the same partition. e.g., in Chrome, JS, HTML Parser, DOM, and Layout belong to one partition (\oplus), while Network and Storage belong to another partition(\circ).</i>	36
4.2	Number of Historical Security Vulnerabilities in Firefox, Categorized by Severity and Firefox Components	46
4.3	Kilo-lines of Source Code in Firefox Components. <i>In our experiments, we consider the following components: 0. NETWORK, 1. JS, 2. PARSER, 3. DOM, 4. BROWSER, 5. CHROME, 6. DB, 7. DOCSHELL, 8. EDITOR, 9. LAYOUT, 10. MEMORY, 11. MODULES, 12. SECURITY, 13. STORAGE, 14. TOOLKIT, 15. URILOADER, 16. WIDGET, 17. GFX, 18. SPELLCHECKER, 19. NSPR, 20. XPCONNECT, and 21. OTHERS.</i>	47
4.4	Number of Calls (in 1000s) across Code Units over 100 Websites <i>This table lists the number of calls from the row component to the column component.</i>	49
4.5	Exchanged Data Size (in kilobytes) <i>This table lists the bytes of data exchanged from the row component to the column component.</i>	50
4.6	Cross-Origin Calls & Sub-Resource Loading	51
4.7	Round-Trip Time (RTT) of Unix Domain Socket, Network and Cross-VM Communications, <i>in nanoseconds</i> , Averaged over 10,000 Runs Each	52
4.8	Security Benefits and Performance Costs of Partitioning Dimensions <i>Performance costs are per page, averaged over Alexa Top 100 websites.</i>	53

5.1	Example Interfaces Defined in BSM Hooks	62
5.2	Summary of Hooks and SLOC	65
5.3	BSM Implementation in Different Browsers	66
5.4	Dromaeo Test Results for BSM	73
6.1	Summary of Case Studies on 3 Popular Web Apps, demonstrating mod- est adoption effort	97
6.2	Estimated Size of TCB in CRYPTON-KERNEL Prototype	97
6.3	Study of 20 Popular Web Applications. <i>< 1% of web application code needs to run in the TEMI to access sensitive info; < 1% of all calls to JS data types trigger browser-TEMI interactions.</i>	99

Chapter 1

Introduction

Web browsers have evolved from a simple document viewer in the old days, to a runtime platform for web applications today. Beyond fetching, processing, and rendering content downloaded from web servers, modern web browsers allow active components from various sources to execute in its execution environment, such as advertisements, JavaScript libraries, web gadgets, and Flash objects. Such a powerful platform enables web applications to provide convenient and attractive services to users in almost all areas once dominated by desktop software, including communications, business transactions, health-care, and even games. A lot of sensitive data are also processed in online transactions, such as credit card numbers, medical records, tax statements, etc. However, the development of security mechanisms in web browsers are not matching the pace of browsers' evolution in functionality. The integrity of online transactions and data privacy are at stake, given the overwhelming number of attacks on the web, and numerous vulnerabilities in web browsers and applications [151, 147]. Existing reactive approaches to browser security fix vulnerabilities discovered by developers or attackers, or propose security improvements to defend browsers against specific attacks. Nevertheless, such approaches do not fundamentally change the situation where security support in browsers is lagging behind the fast evolution in functionality. In this decade, new web features are constantly being introduced and implemented in web browsers, which often give rise to new security loopholes [69, 12]. Instead of existing reactive approach, we advocate more proactive rethink and redesign of web browsers as a new execution platform with principled approaches.

In retrospect, the operating system (OS) is the traditional execution platform that allows different programs to interact with different resources, such as files, network, memory spaces, etc., in the system. Similarly, the web browser platform has also evolved into such a complex environment, supporting web applications from different websites to run and interact with each other. Thus, we study the principles that guide the software design on the OS platform, to better design web browsers. Throughout the years, research on OS security has identified a number of fundamental principles [142] in protecting computer systems, such as *separation of privilege*, *least privilege*, *complete mediation*, as well as often overlooked *economy of mechanism*, etc. Many modern computer systems [131, 18, 57, 22, 25] have benefited from these principles in protect-

ing the integrity of themselves and user data. Unfortunately, such security principles are not well embodied in browser designs. Few initial attempts are limited to protecting browser extensions [13] or isolating resources from different web origins¹ or websites [15, 169, 62, 63]. Nevertheless, these attempts are limited in the scope of threats addressed and security goals actually achieved by the proposed systems. For example, recent browser extension frameworks are found to have failed some of their design goals [96, 86]. Thus, existing solutions lack the fundamental security principles to guide the efforts on securing web browsers. Lack of such security cornerstones leaves the browser platform with loopholes for various attacks to the browser and web applications.

In this thesis, we revisit the fundamental security principles and use them to guide our rethinking of the browser design. One key mechanism we identify in embodying security principles to design web browsers is practical and efficient in-browser isolation of code and data. To be practical, the mechanism should be able to effectively address real-world security threats in browsers, and be as transparent to existing web applications as possible to avoid heavy adoption cost. At the same time, the mechanism should not burden browsers with excessive performance overhead. This thesis proposes new primitives and infrastructures for developing and evaluating such isolation-related mechanisms in web browsers. These proposed solutions address several fundamental challenges during building principled security into the browser platform.

Unlike traditional web applications, where boundaries between distrusting parties can be statically defined, components in modern web applications such as web mashups, JavaScript libraries and advertisements (ads), have made such boundaries blurred and dynamic. These components closely interact with each other. We need flexible and transparent primitives that allow dynamic isolation of untrusted content loaded into web sessions during run time. To maintain compatibility to existing functionalities, such primitives should not only be transparent to web applications, but also to the JavaScript libraries or ads scripts included into the applications. On the other hand, with existing or new isolation primitives, there can be various ways to partition the browser and enforce isolation. In deciding browser partitioning for privilege separation, we need to strike a fine balance between competing concerns, such as security and performance. Privilege-separated designs improve the browser's resilience against vulnerability exploits, yet excessive program partitioning may incur high performance overhead that have prevented systems with desirable security properties from being adopted in practice. We need a systematic approach to evaluate the security and performance implications in privilege-separated designs. This would enable browser designers to identify potential performance bottlenecks in their design, rectify them, or optimize them in the implementation.

Isolation provides desirable security guarantees against attacks exploiting browser vulnerabilities, while modern web applications bring in new challenges at a new level. Web 2.0 applications tend to include resources and active objects, such as JavaScript,

¹An origin is defined as a triplet of the *protocol*, *host name*, and *port* [76].

from multiple sources into the same web origin, some of which cannot be fully trusted. Thus, even within one isolated program partition, we still need mechanisms to monitor and control the run-time behaviors of different running components. Such mechanisms should capture and discern details of sensitive operations and events occurring in web applications, and provide sufficient details for security solutions to detect and pinpoint suspicious behaviors and components among web mashups.

With flexible and effective isolation, access control or behavior monitoring mechanisms, the web browser can constrain and regulate resource access and run-time capabilities from untrusted parties in the web browser. However, in practice, it is difficult to achieve *complete mediation* with isolation and access control, though it is a desirable security guarantee for certain pieces of critical data. Data are often exposed to all code that process or transfer them. Without complete mediation, it is difficult to control information release, since leakage of information is out of control once it is released to any permitted party. The large size of code from different parties running in the browser necessitates centralized and sustained control of sensitive data throughout their lifecycle. Such challenges require new security primitives that provide effective data-centric isolation in modern web applications.

Thesis Overview. We develop solutions to secure web browsers by incorporating fundamental security principles into their designs. Specially, we abstract a security reference model to represent major browser components and their interactions. We use this model to drive the analysis of security support required by modern web applications, and thus develop *a)* a flexible confinement solution for JavaScript-based ads, *b)* a methodology to quantitatively evaluate security-performance trade-offs in browser designs, *c)* a general security framework supporting existing and new security solutions within isolated partitions, and *d)* a new abstraction for sensitive data protection in browsers.

A comprehensive and flexible confinement of JavaScript-based advertisements. We propose an in-browser isolation mechanism to combat attacks from untrusted scripts in advertisements [45]. Attackers may attempt to steal data from web applications or compromise their integrity, via popular attack vectors such as cross-site scripting, cross-site request forgery, prototype hijacking, etc. Further, attackers may compromise the browsers themselves by exploiting their vulnerabilities of buffer overflow, heap spray, dangling pointers, etc. In fact, the latter case is more severe. As most of web browsers today are implemented in C/C++, once an attacker exploits a memory vulnerability, he/she can gain access to web applications inside the victim components as well. To secure web browsers against different web-based attacks, we propose a comprehensive solution to combat threats to both web applications and browsers in a typical scenario among today's web applications: trusted web applications embedding untrusted JavaScript advertisements.

Online advertising has become a core business model of the web. For instance, in

2011, more than 96% of Google’s revenue was from advertising. Driven by considerable financial attractions, popular websites today tend to embed advertisements (ads) into their pages. To select ads contents that are potentially more relevant to the users, on-line ads tend to dynamically decide contents to be displayed according to users’ contexts and profiles, which is called *ads targeting*. Such advertisements are mostly JavaScript-based, so that they can check the contents of the web pages the users are currently viewing. However, when an ad script is included in a web page, it runs in the same privilege as the hosting application. Malicious or compromised ads not only have full access to web application data, but can also potentially exploit browsers’ vulnerabilities to launch attacks such as drive-by downloads. Our solution divides the mixed JavaScript execution environment into two separate JavaScript engines. The scripts from the web applications are executed in the original JavaScript engine accessing the Document Object Model (DOM) normally, while untrusted ads scripts are executed in a separate and sandboxed shadow JavaScript engine, whose accesses to page DOM are tunneled through a policy enforcer. This solution enables isolation-by-default for trusted and untrusted scripts previously executed in the same environment. It also enables flexible policy enforcement to regulate the behaviors of untrusted JavaScript.

A quantitative evaluation of privilege separation in web browser designs. On the other hand, although isolation mechanisms have been widely adopted in enforcing privilege separation and least privilege in various systems including web browsers, little attention has been paid to the other side of the story: isolation of software components also incurs additional performance cost. In practice, privilege-separated designs require a fine balance between security benefits and performance costs. Prior work on privilege separation measures performance overhead “after-the-fact”. In fact, performance overhead has been a main cause that prevents many privilege separation proposals from being adopted in real systems. We thus develop a new measurement-driven methodology that *quantifies* security benefits and performance costs for a given privilege-separated browser design. Our methodology analyzes historical bug database and code size to evaluate security benefits in component isolation, and applies empirical measurements on a browser blueprint over a large-scale test harness to measure performance costs. It then leverages the measurement data to evaluate the security-performance trade-offs in isolation designs among different browsers.

A general security framework for behavior control Within one isolated program partition, or when it is difficult to pre-determine the partitions, we need intra-partition security mechanisms that monitor and control web application behaviors. Such mechanisms also allow security researchers to quickly experiment with different alternatives of program partitioning before implementing it. As such, the browser needs to provide sufficient security support to mediate and regulate sensitive behaviors of web applications, establish finer-grained identities, e.g., intra-origin principals and associate runtime events with these identities. To achieve such goals, we perform systematic analysis

based on a reference model of browser internals, and design a security framework for the browsers [43]. The framework enables *general*, *flexible* and *extensible* support to allow security solutions to monitor and control web application behaviors. We envision this browser security framework would enable a myriad of novel solutions to enhance web security. To demonstrate such capabilities of the framework, we show how our framework can support existing security solutions in browsers, as well as new security solutions that can be built on top of it.

A new abstraction for data-centric isolation. The amount of sensitive data processed on the web platform is increasing steadily. However, as the main execution environment of the web platform, web browsers protect data based on web origins, while not paying special attention to sensitive data that are more critical to their owners. Current web applications have to trust the entire codebase at the client side to protect sensitive data for their users. Such client-side codebase include web browsers, browser add-ons, and web applications, which are known to have many security vulnerabilities. We propose a new data-centric isolation abstraction that bundles sensitive data with permitted operations, controlled by web developers to compute over the data. This abstraction provides strong security guarantees on the integrity and confidentiality of protected data, with a very small Trusted Computing Base (TCB). Our evaluation on 3 real-world applications and a large-scale study on Alexa Top 50 websites demonstrate the practicality and applicability of our approach.

Summary of contributions.

- We design and implement a comprehensive confinement solution `ADSENTRY` for third-party JavaScript advertisements. It sandboxes untrusted ads scripts and regulates all access from them to the DOM of the hosting page, according to a security policy. We evaluate it with all major online ads providers, and demonstrate its effectiveness and practicality.
- We develop a measurement-based methodology to quantitatively evaluate security benefits and performance costs for privilege-separated browser designs. We build an assistance tool `PRIVGAUGE` to automate the measurements, and apply it to study recent browser designs. Our results provide empirical guidelines for future browser designs.
- Based on our systematic analysis on security requirements in browsers, we propose a general security framework `BSM` to monitor and control web application behaviors inside one program partition. We show that it provides the key supports required by existing security solutions, and its potential in enabling new security solutions.
- We propose a new abstraction `CRYPTON` for data-centric isolation in web browsers, which effectively protects sensitive data in various real-world web applications during our experiments.

Chapter 2

Background on Browser Security: Threats & Defenses

In this section, we elaborate the internals of the web browser, and summarize existing attacks against web applications and web browsers. We also discuss existing security solutions against these attacks, and new security requirements raised by modern web applications.

2.1 The Web Browser

A web browser is a key component on the World Wide Web. It is a client-side execution environment that communicates with web servers, executes and renders various contents downloaded from the web to end users. It also receives user interactions, such as keystrokes and mouse movements, and triggers additional communications and content rendering accordingly.

The Mosaic browser, released in early 1993, was considered the first well known graphical web browser. Since then, web browsers have experienced a rapid evolution in its functionality, architecture, and in its applications. Figure 2.1 illustrates major components inside a modern web browser, extracted from Mozilla Firefox and WebKit-based browsers, Google Chrome and the Android Browser.

The Network module handles the communications of the browser with the Internet; the Parser takes in the raw web content received in HTTP responses, and processes it according to HTML format. During parsing, the Parser builds the HTML content into a tree structure called Document Object Model (DOM) that represents the web page. The JavaScript Engine executes JavaScript in the web page and accesses DOM via standard interfaces, and initiates HTTP requests via XMLHttpRequest. The Layout/Rendering engine constructs the rendering tree according to DOM, and renders the pixels onto the screen via system libraries or GPU commands. The Browser Event Manager is a conceptual component that handles event registration, and invokes registered event handlers when events occur. As modern browsers are event-driven software, in reality, the functionality of event management spreads over all major browser components.

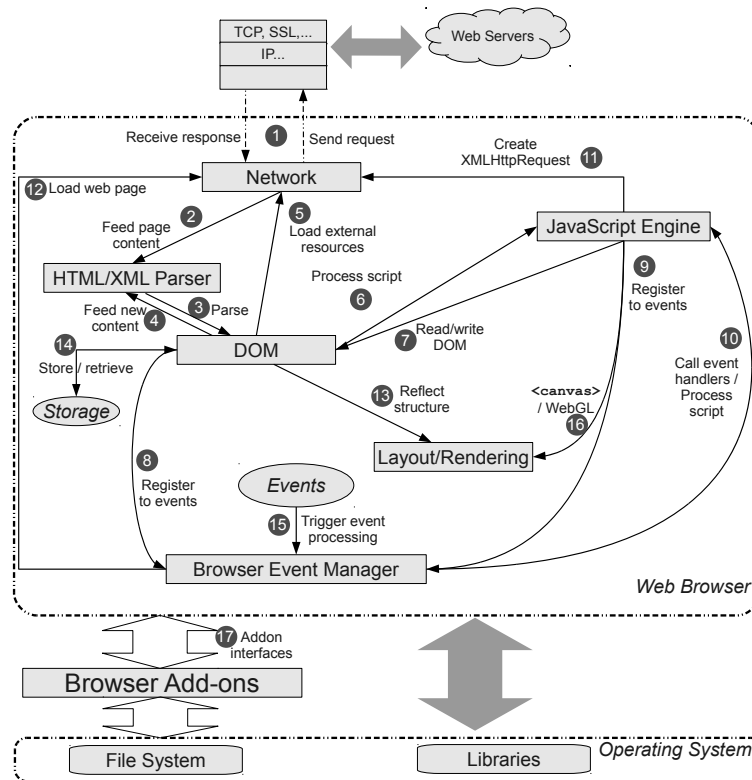


Figure 2.1: Browser Internals

Rectangles denote major components in the browser, dotted lines denote interactions between components, and arrows represent the directions of interactions. Circled numbers denote the categories of inter-component communications.

Browsers also provide interfaces for application developers to write add-ons to enhance the functionality of browsers.

Now we will use typical web browsing scenarios to illustrate the interactions between major components in the reference model. Normally, a user loads a new web page by starting the browser with the pre-configured homepage, or by loading a new page entered in the address bar, or by selecting an entry in the bookmark or browsing history. Such user input behaviors are processed by the Browser Event Manager. According to the different protocols of the resource specified by the user action, the Browser Event Manager in turn invokes the Network module to load a web resource (Category 12 as in Figure 2.1), or sends the script to the JavaScript engine for execution (Category 10).

The Network module handles the transport layer communications (Category 1) with web servers. When it receives the raw web content from the network, it passes them to the Parser (Category 2).

The Parser in turn parses the raw web content according to its encoding and content type. If the content is in HTML, it parses the HTML content and builds it into the Document Object Model (Category 3).

When elements that require additional resource requests are inserted into DOM, such as HTML image elements, HTML script elements, or iframes, the DOM module

invokes the Network module to load them (Category 5). When the inline script elements are added to DOM, or third-party scripts have been loaded into DOM, the DOM module sends the script to the JavaScript Engine for execution (Category 6).

On the other hand, JavaScript can also dynamically modify DOM (Category 7), by adding/updating elements or attributes, or by inserting new fragments of page content. After that, newly introduced page content is sent back to the Parser for parsing (Category 4), as such new content may also contain JavaScript that inserts another new script. In this thesis, we use DOM to refer to both the standard DOM specification [166] and conventional Legacy DOM or DOM Level 0 interfaces [172] implemented in most modern browsers, such as the *window* object.

We do not include a separate Layout Engine in this reference model, as it is not directly accessible by web applications. In our security reference model, we consider the layout as part of the DOM module that handles the UI rendering.

The JavaScript Engine also registers to certain events (Category 9) upon the demand of running scripts. Similarly, HTML elements can also specify event listeners via attributes such as *onclick*, *onmouseover*, etc. This is handled by the DOM module in our model (Category 8). When the registered events occur, the Browser Event Manager invokes their corresponding event handlers registered earlier (Category 10).

The JavaScript Engine creates XMLHttpRequest, invokes the Network module to send it out (Category 11), and receives the response from event listeners (Category 10).

The Layout and Rendering engines construct the rendering tree according to elements and attributes in the DOM tree (Category 13). Recently proposed features `<canvas>` [164] and WebGL [87] allow web applications to directly render a particular region using JavaScript (Category 16).

Local storage and session storage interact with the page DOM via interface Category 14.

We model all behaviors related to event registering and handling in the Browser Event Manager (Category 15).

Browser add-ons interact with the browser components and data using interfaces provided by web browsers (Category 17).

The primary security model in browsers: the same-origin policy (SOP). A single browser instance can load web contents from various sources. To prevent data leakage from one source to another, web browsers enforce the same origin policy (SOP) [113]. Under SOP, resources from one origin can only be accessed by JavaScript from the same origin. SOP establishes the corner stone for browser security, although it has various insufficiencies as discussed below.

1. *The inconsistent enforcement of SOP under different scenarios.* As documented in detail by the Browser Security Handbook [183], SOP is enforced differently for different resources. For example, cookie access is subject to the values of `document.domain` set by the target domain, and XMLHttpRequests can be

used to communicate with other origins with the Cross-Origin Resource Sharing (CORS) [165] technology. Such inconsistency in resource access restrictions causes it difficult to protect all resources in the browser [149].

2. *Third-party resources included.* When SOP was first proposed in late 1990s, the division between different web origins was much clearer. However, in today's web applications, active components are frequently included from third parties, and they are executed in the origin of the hosting page. Under such scenarios, origin-based security seems too coarse-grained, and is insufficient for preventing attacks that occur right inside the victim application's origin.
3. *Limited to web application protection.* SOP is only enforced on contents from web applications. Browser code, either in native languages, or in JavaScript, is not subject to SOP. Besides, the correct enforcement of SOP requires the assumption of the sanity of relevant browser code. However, this assumption can be broken by attacks against vulnerabilities in the browser code itself, which has many.

As we can see that although SOP lays down a useful foundation to protect data across origins, it is far from providing comprehensive protection to web applications and browsers. Next, we overview popular web-based attacks that threaten the web clients.

2.2 Threats to Web Clients & Existing Countermeasures

Attackers can launch web-based attacks targeting vulnerabilities in any portion of the web clients, including those in web applications, web browsers, operating systems, and even hardware. In this thesis, we focus on web-based attacks against web application and browser vulnerabilities.

2.2.1 Threats to Web Applications & Existing Defenses

Malicious JavaScript. The most dangerous attacks to web applications are caused by malicious JavaScript. Under the same origin policy, once a piece of malicious JavaScript is included in an origin, it can send requests to the web application server with the current user's privilege, access web applications' data and disclose sensitive information. It can also include new scripts at runtime or modify other scripts in order to change the behaviors of web applications.

- Cross-site scripting (XSS). In this attack, malicious JavaScript is injected into victim web applications. According to same-origin policy, the injected script is granted with the privilege of the victim origin.
- Malicious mashup, JavaScript advertisement or library. A mashup website combines content from more than one source. To enable interactive and responsive

user experiences, web applications commonly include JavaScript from various untrusted sources, such as, third-party JavaScript libraries, and mashups. They may also include JavaScript ads for profit. All embedded components in a web page run with the privilege of the integrator, so malicious components can directly access the integrator's data and tamper with its execution.

Defense. To prevent XSS attacks, a common idea is to distinguish different pieces of JavaScript in a web application and enforce policies on them [67, 83, 150]. For example, Noncespaces [67] appends random prefixes to HTML tags at the server side, and detects injected tags at the client side since they would not have the correct prefixes. The Content Security Policy [150] allows web applications to specify the list of sources where JavaScript, images, or iframes can be loaded from. Noxes [88] is a lightweight application-level firewall that blocks suspicious XSS attempts with rules specified by users or inferred automatically.

These solutions need support from the web browser to fetch and enforce policy on the received HTML content by intercepting HTTP response processing (Category 2 as in Figure 2.1) and the parsing of HTML content (Category 3).

For mashup web applications, another line of solutions [82, 168, 37] regulate JavaScript's accesses to resources. For example, ESCUDO [168] employs a ring-based security model to enforce that JavaScript can only access DOM content of less trust. MashupOS [168] proposes new primitives to allow web applications to choose different privilege models for third-party contents with different trust. ADJAIL [158] isolates untrusted JavaScript-based advertisements in iframes, and regulates their access to the hosting web page content by tunneling its DOM data via messages to the iframes.

Such solutions require the browser to distinguish and track scripts in a web application and regulate their access to browser resources. To implement these solutions, various components in the web browser need to be intercepted, including HTML content parsing (Category 3), and JavaScript's accesses to DOM and XMLHttpRequest (Category 7, and Category 11).

Another category of solutions on malicious JavaScript tends to restrict its functionality with a safe subset of the JavaScript language [38, 156, 50, 79, 52]. These solutions allow rewriting of untrusted JavaScript into a subset with reduced resource access and language functionality. They have been applied in real-world applications to prevent untrusted JavaScript from compromising the hosting pages.

Prototype hijacking. JavaScript engine is an active component in browser reference model, which provides standard function objects to JavaScript during execution. JavaScript allows overriding of function objects at runtime. In this attack, malicious script exploits this feature to overwrite standard global JavaScript objects or objects of other scripts. This way, it subverts the behavior of the victim web application.

Defense. To protect the environment modification by malicious script, JCShadow [124] partitions JavaScript code in a web page into multiple groups and isolates JavaScript

objects of one group from another. To implement this solution needs intercepting access to JavaScript object inside JavaScript engine.

Other malicious requests. HTTP requests expose the APIs of web applications to clients. In the complex browser environment, malicious requests can also be generated without JavaScript.

- Cross-site request forgery (CSRF) [14]. In a CSRF attack, a malicious website or malicious content embedded in another website automatically sends or tricks the user into issuing a malicious request to a web server where the victim user has already logged in. The reason CSRF attacks can succeed is that web browsers automatically attach users' session cookies to outgoing HTTP requests.
- Clickjacking [70]. Clickjacking is a malicious technique that overlays a victim web page in a transparent frame over an attacker's web page. It aims to trick web users into clicking on victim web page to unwittingly perform actions, such as deleting emails, transferring funds, etc.

Defense. To prevent CSRF attacks, a class of research solutions [39, 84, 101] are proposed that focus on enforcing security policies at the client side (Category 2, and Category 5). Other research works [14, 85] aim to prevent CSRF by sending additional information in the request to the web server (Category 5, and Category 11).

To prevent clickjacking, ClickIDS [10] introduces a solution to automatically detect clickjacking attacks by intercepting user events (Category 10) and retrieving all elements when a click occurs (Category 7). If at that time more than one click-able element is under the mouse pointer, it shows an alert to users.

In summary, defenses to this type of attack require understanding the execution environment of web applications to decide the legitimacy of runtime behaviors.

Privacy violation. Since browsers are the shared environments of multiple web applications, privacy violation attacks on user data and behavior are possible [81].

- History sniffing. In history sniffing attacks, the attacker can use, for instance, JavaScript or requests to CSS background to determine the color or style of links to discover which sites the user has visited.
- Behavior tracking. This attack allows website to track users' keyboard or mouse behaviors when they visit the web sites with tracking facilities, which may violate users' privacy without informed consent.

Defense. There have been several studies on privacy violation behaviors [81, 161]. To capture or prevent history sniffing, it requires mediating accesses to properties of CSS or DOM elements by JavaScript (Category 7). To monitor or prevent behavior tracking attacks, it needs tracking of the principals and JavaScript's accesses to DOM and events (Category 7, and Category 9). The principal of the script needs to be checked against

the principal of the DOM element or event being accessed to decide whether to allow or deny such accesses.

2.2.2 Threats to Web Browsers & Existing Defenses

The web browser as a software is also susceptible to traditional vulnerabilities such as buffer overflow [119], integer overflow [121], heap spray [42], dangling pointer [6], etc. According to our study, such vulnerabilities amount for around 76.5% of historical security vulnerabilities of Firefox throughout the past years [55].

Defense. Current countermeasures taken by most browser vendors are to patch these vulnerabilities whenever detected by testing, exploited by attackers, or reported by users.

More systematically, researchers have been applying privilege separation into browser designs [36, 169, 15, 75, 154]. The basic idea is to divide the browser into multiple partitions, each isolated from each other, using processes, or even virtual machines. Under such isolation, each partition can only be granted with the least privilege in accessing other partitions or system resources. Privilege-separated designs can mitigate the damage caused by exploits since the attacks will be contained in the victim partition. We provide detailed analysis and measurements on these proposals in Section 4.

2.3 New Security Requirements from Modern Web Applications

Existing defenses can defeat a large portion of prevailing attacks against web applications and browsers. However, they are not sufficient, especially with the new challenges coming from modern web applications aggregating contents from multiple parties.

Stronger and more flexible isolation mechanisms. Mixed contents in the same web application call for intra-origin isolation mechanisms that allow web developers to render untrusted contents in a controlled environment. Nevertheless, the comprehensiveness and flexibility of existing intra-origin isolation mechanisms are not satisfactory. Solutions such as MashupOS [168] and ADJAIL [158], mainly focus on access to DOM data. However, untrusted contents are not constrained from attempting to exploit the browser codebase, such as the JavaScript engine that executes the untrusted JavaScript. On the other hand, maintaining compatibility for legitimate access is also important. For example, we need to keep the original orders of execution of different scripts on the hosting page to avoid unnecessary alternation of web application logic. We also need to have the capability of dynamically regulating access from untrusted contents to trusted contents. Existing isolation mechanisms are short of either comprehensiveness or flexibility for modern web applications with heavy content aggregation.

Security and performance implications of privilege partitioning in browsers. Decades ago, web pages in the browser are displayed as separate entities without much interac-

tion between each other. Today, different web contents from different sources have close interaction in the browser, in terms of sub-resource loading, `postMessage` communication, script inclusion, etc. Privilege-separated browser designs that divide browser components and resources into different partitions will inevitably cause performance overhead for inter-partition communications. Such performance overhead can amount to a tremendous scale with excessive partitioning. Prior practices of privilege separation tend to evaluate performance implications after a certain implementation has already been developed. However, in practice, even if the performance is unsatisfactory, it could be very difficult and costly to change the redesign and re-implement the system. A more practical solution would be to evaluate potential security and performance implications during the design phase, and determine their tradeoffs before the implementation.

General support for flexible control of web application behaviors The newborn Web 2.0 has revolutionized how web applications are built. The boundaries set by origins are blurred in modern web applications, where resources from different origins are executed in the same origin of the integrator. To address threats within such new environments, security solutions need to have a deeper understanding of the run-time accesses to resources from web applications, and accurately identify the application code or events that initiate the access. Reasonable security decisions can only be made with such detailed information, instead of blindly blocking or allowing certain types of access for all application code. Once built into the browsers, such general security support will enable various security solutions to be deployed without the need of directly modifying the browser.

Protecting sensitive web content with data-centric isolation. With more sensitive corporate and user data being migrated to the web platform, it becomes crucial to ensure data security on the web. However, modern browsers are complex systems with numerous components, principals, and convoluted interactions between these components and principals. Moreover, tons of security vulnerabilities exist in web browser and application code. Existing efforts in securing the browser either fail to achieve *complete mediation* in such complex browser code base (such as access control), or lose control of the information once it is released to any party (such as information flow tracking). Considering the critical business values or personal privacy behind sensitive information, it is desirable to have new primitives that provide web applications with strong data protection, without trusting a large client-side codebase.

Chapter 3

ADSENTRY: Comprehensive and Flexible Confinement of JavaScript-based Advertisements

Internet advertising is one of the most popular business models of today’s Internet companies. For example, in 2011, more than 96% of Google’s revenue was from Internet advertising [61]. In Internet advertising, website owners or web publishers include advertisements (or “ads”) from advertisers in their pages, and get paid by advertisers when users view and click on these ads.

To increase the likelihood for users to click on the ads, advertisers commonly use (JavaScript) code in ads to check a user’s browser environment to select advertisements that are believed to be more attractive to the users. As third-party code, these ads unfortunately pose great security threats to both web applications and the underlying operating systems. For example, such ads require close integration with the displayed page contents, which may leak users’ private data [102] and break web applications’ integrity. Worse, malicious ads can further exploit software vulnerabilities in web browsers to launch drive-by downloads and surreptitiously install malware on users’ machines. A recent research shows that “about 1.3 million malicious ads are being viewed online every day, most pushing drive-by downloads and fake security software” [115].

To mitigate the threats from untrusted ads, a number of solutions have been recently proposed. They address the threats to user privacy and web application integrity by sandboxing JavaScript ads through functionality restriction or isolation [38, 156, 158, 50, 52, 64, 99, 98, 79, 125, 135, 180, 105, 168]. However, they cannot block ads from triggering drive-by downloads, which have been “persistently” plaguing online users as one of the main attack mechanisms. In addition, these solutions are not flexible in controlling behaviors of JavaScript advertisements: the allowed access of an ad must be decided before it starts to run. In the face of these limitations, there is a need for an integrated solution that can not only flexibly regulate the ad access to various web contents, but also effectively block drive-by downloads from malicious ads.

In this work, we present the design, implementation and evaluation of ADSENTRY,

a comprehensive and flexible isolation framework to confine JavaScript-based advertisements. Instead of supporting only a subset of JavaScript functionality or isolating the ad execution through significant changes to web pages, ADSENTRY provides a *shadow JavaScript engine* for untrusted ad execution. The purpose of having a shadow JavaScript engine is to ensure that the ad will not affect the host web content without proper control, thus protecting user privacy and the integrity of web applications. More importantly, it provides the control in a transparent manner and still exposes the full spectrum of JavaScript functionality to the untrusted ads. Meanwhile, to block possible drive-by downloads and preserve the integrity of the host system, the shadow JavaScript engine is strictly sandboxed.

By design, ADSENTRY effectively mediates all accesses made by untrusted ads to the web application. We stress that the shadow JavaScript engine (for the ad execution) by default cannot access the original page DOM (Document Object Model). To accommodate legitimate accesses by ads to some part of the page content, ADSENTRY transparently interposes related DOM accesses from the ads. For every such access, ADSENTRY checks its legitimacy (according to a given access control policy) and, if benign, redirects it to the original page DOM to substantiate the access. Our framework is flexible in allowing both web publishers and end users to specify or customize the access control policies for ads, as well as allowing dynamically changing policy after an ad starts to execute.

We have implemented a proof-of-concept ADSENTRY prototype. The shadow JavaScript engine implementation is based on the open-source Mozilla SpiderMonkey. Its execution is strictly sandboxed with Native Client [179], which has demonstrated its effectiveness in confining third-party code with high efficiency and reliability. Our development experience further indicates that ADSENTRY is a generic framework that can be conveniently implemented as a regular browser extension without requiring the modification of the browser code. Our evaluation results with a number of ad-related exploits show that ADSENTRY is effective in successfully blocking all of them. The performance evaluation shows that the protection is achieved with a low overhead.

3.1 Problem Overview

In Internet advertising, advertisers pay web publishers directly to display their ads on these websites, or more often, pay advertising networks to get their ads displayed on popular sites, easily reaching out to a large amount of audiences. Moreover, advertisers usually allow web sites or advertising networks to dynamically decide what kind of ads to display to their visitors, based on the web contents users are viewing. This behavior is called “targeting” of ads. It makes Internet advertising more relevant and presumably more helpful to visitors. The profit of web publishers hosting ads can be calculated by different revenue models, including measuring how many times the ads are displayed, how many visitors have seen the ads, or how many times the ads have been clicked by visitors, etc. [173].

Internet advertising brings in new challenges to web security and privacy. One possible way for publishers to include advertisements is to completely isolate them in separate iframes. However, such a complete isolation makes advertisement targeting impossible. As a result, third-party ads are often included in `<script>` elements, so they have the same privilege as other JavaScript on the web page.

In this work, we focus on such third-party JavaScript-based advertisements (ads) that are deployed on web pages. These ads are hosted outside web publishers' servers, but included as JavaScript on the web pages. If some of them become malicious, they may abuse their privileges in accessing web application data for various purposes, such as leaking confidential user information and issuing unauthorized transactions. Moreover, they may exploit software vulnerabilities in browsers to take over the users' systems.

Our goal in this work is to comprehensively confine these untrusted JavaScript ads and effectively protect users' privacy and the integrity of both web applications and the users' computer systems.

3.2 System Design

To effectively confine untrusted ads, we have four design goals, i.e., *comprehensiveness*, *flexibility*, *transparency*, and *efficiency*. By comprehensiveness, we aim to provide an integrated scheme that not only regulates ad access to the host web page, but also contains malicious ads from launching drive-by downloads. The flexibility requirement allows both web publishers and end users to specify access control policy for ads. Users can also dynamically change access control decisions based on application run-time states. The transparency goal requires no modification to the browser for the support and preserves the timing of the JavaScript behaviors in the web applications. It also requires that the current billing model of ads is not affected, e.g., in the number of user clicks and impressions. Also, the proposed solution needs to be efficient in introducing low performance or maintaining a similar level of user experience.

Following these design goals, we have developed a novel ad isolation framework called ADSENTRY, whose overall architecture is shown in Figure 3.1. In essence, ADSENTRY provides a shadow JavaScript engine to confine untrusted ads. This shadow JavaScript engine by default has no direct access to the original browser environment and the operating system. Therefore, the ads can be fully confined, and the host web page and OS will remain intact even if the ads are malicious. To meet the transparency requirement, the shadow JavaScript engine can be seamlessly integrated into current browsers through the standard browser's extension application programming interfaces (APIs), i.e., no browser modification will be necessary.

With the introduction of a shadow JavaScript engine, ADSENTRY essentially works with two JavaScript engines: the untrusted ads run inside the shadow engine while the rest (normal) JavaScript in the web page runs as usual in the default engine. Untrusted ads can be specified by web publishers and end users, or automatically identified by

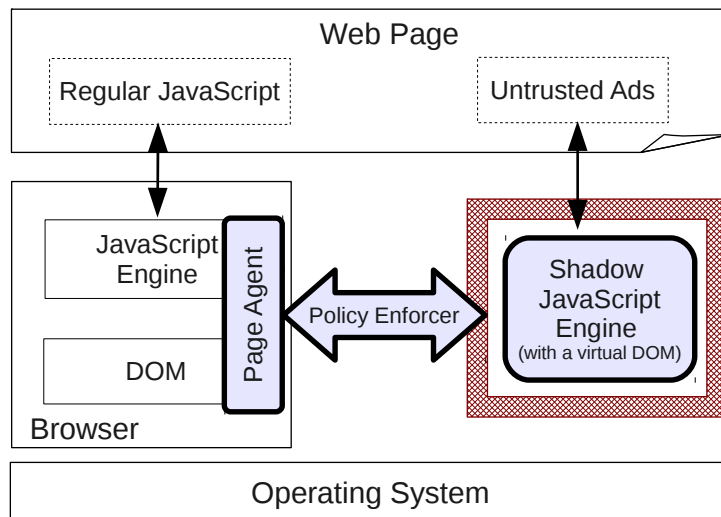


Figure 3.1: An Architecture Overview of ADSENTRY
The core components of ADSENTRY are highlighted in the figure.

tools, as we detail in Section 3.3.1. We point out that an ad may have legitimate reasons to access certain web content (e.g., for the purpose of advertisement targeting). To accommodate these requests, ADSENTRY provides a virtualized *DOM* to the shadow JavaScript engine. The virtual *DOM* has all the standard *DOM* interfaces, including `XMLHttpRequest`, so page accesses made by ads running in the shadow engine will be received by the virtual *DOM*. When the virtual *DOM* is being accessed, it will relay the access to the *page agent* in the browser through a *policy enforcer*. The policy enforcer will decide whether a page access is allowed by users' security policies. If yes, the page agent proceeds with the access request on behalf of the isolated ad, and returns the results back to the isolated ad through the virtual *DOM*. If not, the access will be blocked to protect the integrity of the web page.

Besides virtualizing the *DOM* access for untrusted ads, ADSENTRY also sandboxes the ad execution within the shadow engine, preventing them from compromising users' operating systems.

It is important to note that ADSENTRY is transparent to web pages by automatically dispatching ads to the shadow engine and seamlessly supporting their accesses. As a result, the billing model of ads is not affected. ADSENTRY preserves the original execution timings of all JavaScript in the web page, including ads scripts running in the shadow JavaScript engine. This is a key advantage of ADSENTRY over *iframe*-base isolation techniques, such as AdJail. This ensures that the behaviors of the applications and user experience will not be altered unless for security concerns, making ADSENTRY applicable to securing a wider class of untrusted JavaScript code in web applications.

3.2.1 Shadow JavaScript Engine

By introducing a shadow JavaScript engine to host untrusted ads, ADSENTRY allows us to achieve the comprehensiveness goal: resilience against exploits to browsers themselves and protection for the confidentiality and integrity of web application data. As mentioned earlier, the shadow JavaScript engine is executed inside a Native Client (NaCl) [179] sandbox. Our solution is not dependent on any specific sandboxing mechanism. The requirement is just to restrict the program's access to system resources. We choose NaCl for two practical reasons. First, NaCl sandbox has been shown to be secure against code injection attacks with minor performance overhead. Second, NaCl has been supported on a number of platforms, such as x86, x86-64 and ARM [148], which can be very helpful for adopting ADSENTRY by end users, especially with the rising popularity of the Google Chrome browser and the Chrome OS that ships NaCl as one built-in component.

Like the normal JavaScript engine in the current browser, the shadow JavaScript engine is shared across web pages in the browser. To distinguish different ads from different pages, each ad will be assigned a unique identification number. With that in place, when an ad needs to be executed, ADSENTRY sends its JavaScript and the ad's identification number to the shadow engine. To preserve the original execution timing of the web page, the browser waits until the ad's JavaScript finishes in the shadow engine, in the same way that the original browser JavaScript engine handles its execution.

Specifically, once the sandboxed JavaScript engine receives a JavaScript to execute, it creates a new JavaScript context for the ad with the associated identification number. A virtualized DOM will be initiated to contain a set of global objects, which are then made accessible to the JavaScript context. The virtual DOM has all standard DOM interfaces, but each interface is simply a stub that forwards the access to it to the page agent in the browser. After the initialization, the shadow JavaScript engine starts executing the received JavaScript. If the script accesses a particular DOM interface, the access is intercepted by the virtual DOM, which in turn communicates with the page agent to handle the access request.

3.2.2 Page Agent

The intercepted DOM access requests from the virtual DOM are forwarded to the page agent, which resides on the same page with the web application. After the verification from the policy enforcer, the page agent will perform the requested DOM access on behalf of the ad. If the request is to create or modify DOM element(s), the page agent takes special care to capture all resulting JavaScript executions and forwards them back to the shadow engine for processing. For instance, when a user clicks on a button created by an ad, the triggered `onclick()` function call will be captured and executed in the shadow engine.

The communication between the virtual DOM and the page agent is in the form of message passing. When the page agent receives a message requesting a DOM access

from the shadow JavaScript engine, it extracts the access from the message, and processes it in the context of the original web page. The page agent then sends the result back to the virtual DOM, and in turn, to the shadow JavaScript engine, completing the access made by the confined ad.

ADSENTRY also naturally regulates access to HTTP requests from untrusted ads. Specifically, ads may initiate HTTP requests by either generating new DOM elements (that have already been controlled by the page agent), or by directly initiating XMLHttpRequest. As XMLHttpRequest is not part of the JavaScript engine, but is provided by the virtual DOM, the invocation to XMLHttpRequest is also regulated by our system.

In order to ensure that the relayed DOM access is transparent to the executing ad, we need to address a few issues – some of them come from innate JavaScript features.

Dynamically generated JavaScript. Ads can insert a new piece of JavaScript into a web page. The new JavaScript must also be executed in the same shadow JavaScript engine. Otherwise, the newly generated JavaScript can escape the isolation of ADSENTRY.

There are several ways for ads to introduce new JavaScript into the original web page. Examples include abusing `document.write` or setting the `innerHTML` attribute of an element. Accordingly, whenever ADSENTRY receives a message from the shadow JavaScript engine requesting to invoke such DOM interfaces, the request is interpreted and all newly introduced JavaScript is properly flagged to ensure they will execute in the shadow JavaScript engine. We detail this solution in Section 3.3.

Timers and event listeners. One interesting challenge comes from the support of asynchronous events, such as timers, where ads register callback routines to be executed later. Such callback routines need to be executed in the shadow JavaScript engine. To handle these asynchronous events, ADSENTRY dynamically creates a stub as the corresponding event handler in the web page. This stub will notify or invoke the true callback routine in the shadow JavaScript engine.

Unfortunately, as asynchronous events occur unexpectedly, the notifying message sent to the shadow JavaScript engine may arrive in the middle of the execution of some other DOM accesses, which causes an undesirable race condition. To avoid that, the messages from asynchronous events will be separately marked and temporally buffered by the shadow JavaScript engine. These messages will then be processed after the ongoing DOM accesses are finished.

Anonymous functions. The JavaScript language supports anonymous functions. For example, the following code snippet creates an anonymous function with a function body `alert(0)`.

```
window.addEventListener("click",
    function () { alert(0); },
    false);
```

As an ad may create these anonymous functions, we need to isolate them properly. Particularly, if these anonymous functions are being used as event listeners, they should be invoked within the shadow JavaScript engine when the corresponding events occur. Unfortunately, anonymous functions are represented as native function objects in the JavaScript engine, rather than strings of JavaScript code. Therefore, we cannot handle them in the same way as we do for JavaScript code on the page.

To address this problem, in our system, when the shadow JavaScript engine executes a statement that creates an anonymous function, we record the function's internal identification number, associate that number with the related DOM access, and forward it to the page agent. When the page agent receives the message at the real DOM side, it dynamically composes a new JavaScript function whose task is just to send a message containing the identification number of the anonymous function to the shadow JavaScript engine. After that, it assigns the newly composed function as the argument to the event listener. When the event occurs, the newly composed function will be invoked (at the real DOM side) to send a message to the shadow JavaScript engine and ask it to run the anonymous function with the specified identification number.

3.2.3 Policy Enforcer

By confining the shadow JavaScript engine within a sandboxed environment, our system effectively blocks possible drive-by downloads that target the underlying JavaScript engines (more concrete examples will be shown in Section 3.4). In the meantime, it is important to point out that the sandbox itself does not provide any guarantee on the confidentiality or integrity of the web application. As a result, it needs to work in concert with the policy enforcer to achieve this goal. Specifically, the policy enforcer checks the requests intercepted by the virtual DOM according to a given user security policy. Only if allowed by the enforced security policy, the request will then be forwarded to the page agent for processing.

As mentioned earlier, our system allows both web publishers and end users to customize the access policy for ads. Specifically, for web publishers, as they can simply change the web page content, they may choose to wrap the ad and confine its execution in the shadow JavaScript engine. For end users, our current system leverages Adblock Plus [123] to automatically identify ads and confine them with a customized JavaScript wrapper. We will present the details as well as the supported policies in the next section.

3.3 Implementation

We have implemented a proof-of-concept prototype of ADSENTRY based on the browser extension support of Firefox, and it is implemented and tested in Mozilla Firefox 3.5.8. Our implementation of the shadow JavaScript engine is based on Mozilla SpiderMonkey version 1.8.0. The virtual DOM code has around 3595 SLOC, which is generated offline with a code generator of 770 SLOC in perl. On the browser side, the other two components (i.e., the policy enforcer and the page agent), are implemented entirely in

the JavaScript language. These two components add about 3100 SLOC.

3.3.1 Specifying Advertisement Scripts

ADSENTRY is flexible in deployment. It allows both web publishers and end users to specify the scripts to be executed in the sandbox. The intuitive way is to associate an attribute with the script indicating it is an advertisement script. However, this solution requires the browser to be modified to recognize the attribute. In ADSENTRY, we provide a function `sandboxAds`. It takes the body of an ad or the URL of an ad script, and notifies ADSENTRY to execute it in isolation.

Therefore, to use ADSENTRY, web publishers can process the advertisement with the function `sandboxAds`, as illustrated by the following example, where the last argument indicates whether the first argument is the URL (`true`) or the body of an ad script (`false`).

```
<script>
  sandboxAds('http://ads.com/ad.js',
            id, true)
</script>
```

ADSENTRY also provides the option to end users by automatically identifying ads instances at the client side. It uses Adblock Plus [123] to identify ads and automatically processes them with `sandboxAds`.

3.3.2 Shadow JavaScript Runtime and Virtual DOM

We use NaCl to sandbox the SpiderMonkey JavaScript engine in a separate process from the original browser process. The shadow JavaScript engine communicates with the original browser process via Unix pipes. For convenient deployment, we develop a Firefox extension that dynamically spawns the NaCl-sandboxed process at run time for untrusted ads scripts. We found this step relatively straightforward to implement.

The main challenge comes from the extension we make to the original JavaScript engine. Specifically, to enable ads running in the JavaScript environment to access related page content (e.g., for ad rendering), there is a need to provide virtual DOM objects. In our prototype, a virtual DOM is made available to the JavaScript engine in the form of a tree of objects. The root of this tree is called the global object.¹ In the case of a web page, the global object is the `window` object. This global object has a number of properties, including global JavaScript variables and functions, such as the `document` object, the `location` object, and the `eval` function. With this tree structure, all other virtual DOM objects are also properties of their parent objects.

We obtain a standard DOM structure from the standard DOM specifications [166], construct virtual DOM objects and expose them to the shadow JavaScript engine as host objects. More specifically, the virtual DOM for SpiderMonkey is generated in

¹Note that the concept of the global object here is different from that of global objects in a JavaScript program.

the following steps: 1) Create a new `JSRuntime` object and set up initial configurations and in runtime, create `JSContexts` for the execution of ads scripts. 2) Create a `JSClass` for each class of DOM objects. 3) Specify properties and member functions for each `JSClass`. 4) Implement property and function accessor methods for each `JSClass`, most of which will invoke one of the centralized access handling functions, respectively. 5) Implement the centralized access handling functions for virtual DOM accesses. These functions will then relay the access to the page agent (on the browser side). To relay the access, they also perform other tasks, such as preparing arguments to actual DOM function calls, looking for anonymous functions, buffering event listener code for later execution, etc. These functions interpose each and every access from ads to the real DOM. 6) Create instances of standard objects from `JSClass` definitions, starting from the `window` global object. For non-global objects, we will specify their parent objects during the creation to form the tree structure.

Considering the large number of virtual DOM objects we need to construct and the associated tree structure we need to maintain, we have a code generator in place to automate the above steps for all virtual DOM objects. The code generator reads in an XML file that specifies the DOM tree objects and structures, and then generates an output file that embeds the JavaScript engine and sets up its host environment with the virtual DOM.

3.3.3 Page Agent

To facilitate the communication between the shadow JavaScript engine and the page agent, we define a simple message format for data exchange. The format is summarized as follows:

```
msg ::= command data
command ::= script | callFunc | getProp
          | setProp | return
data ::= <text>
```

Each message contains a *command* field and a related *data* field. Our prototype has defined five different commands: a `script` command is used to notify the page agent that an ad script needs to be sent to the shadow engine for execution. Upon receiving the message, the shadow engine will prepare the runtime environment and then start executing it. During execution, it will intercept any DOM access from the ad script and based on the type of access, translate it into three other types of messages to the page agent: `callFunc` for function invocations, `getProp` for property retrievals, and `setProp` for property (re)initialization. Finally, a `return` command carries the results in the message body, i.e., the *data* field.

The page agent extends the Firefox browser through its standard extension interfaces. We create a Firefox extension, which monitors the dispatched message events notified by `sandboxAds`. Specifically, following the above message format, if a *script* command is received, it parses the message stored in the event object, and communicates with the sandbox. We stress that web pages cannot directly communicate with the

sandbox, and all communications are done via the Firefox extension. During the ad execution, if it needs to access a DOM object, the sandbox intercepts it and encapsulates the access by sending a message to the page agent requesting a DOM access. Here, the DOM access is meant for the access of the real web page and the extension cannot evaluate it in its own execution environment.

There are two possible approaches for our extension to evaluate the intended DOM access in the web page context. The first approach is straightforward: simply posting a message (or dispatching a custom event) to the web page. After receiving it, the web page can then evaluate the requested DOM access (encoded in the message or event). However, message passing is asynchronous, which allows other JavaScript on the same web page to preempt the execution of the current ad script. This kind of preemption may cause serious problems as it alters the original execution order of different scripts on the page. For example, scripts may have dependency on each other, and a premature execution of a later script may fail if the dependent script has not been executed. As another example, `document.write` is normally executed before a web page is loaded. If it is executed after a page is loaded, it creates a new page, completely eliminating the original one. To execute sandboxed scripts normally, ADSENTRY should not alter the original execution order of scripts on the web page, so this first approach is not suitable here.

The second approach is to implement the communication between the web page and the extension like a function call. Mozilla Firefox provides a mechanism for extensions to evaluate JavaScript code in web pages' privileges, called `evalInSandbox` [109]. In our prototype, we leverage this method to call a function in the context of the web page that contains the ad script, which in turn evaluates the DOM access being requested, and returns the result to the extension. After that, the extension sends it back to the sandbox via a pipe. By doing so, when an ad script is being executed, we can ensure the JavaScript engine in the original browser environment is always in one of the three states: a) waiting for messages from the sandbox; b) executing our script in the extension; or c) executing the message processing function in the host web page while our extension is waiting for the return. As a result, no other scripts on the web page could preempt the current execution of ads script.

Consequently, our implementation is based on the second communication approach. More details are discussed below.

Multiple ads scripts. ADSENTRY supports processing multiple ads scripts at run time. To avoid mix-ups of ad scripts from different web pages, our browser extension maintains a message queue to ensure that only one ad script is being processed at any point of time. Each message sent to the shadow engine is marked with an identification number, enabling the engine to evaluate each ad script in its own JavaScript context. When evaluating DOM accesses requested by the sandbox, ADSENTRY also makes sure the accesses will be evaluated in the same page that originally contains the ad script being executed.

Object maps. The communication mechanisms implemented in ADSENTRY are text-based, but in some cases we need to pass objects as parameter or return values. This is achieved by maintaining object maps at both the page agent and the shadow JavaScript engine, and only communicating the objects' indices in the messages. Before a JavaScript object is to be communicated to the other end, it is checked against the local object map. If it already exists in the map, its index is returned; otherwise, it is inserted into the map with its new index returned. Then in the message sent, the index of the object is included, rather than the object's real data. Next time when a message is received from the other end containing an object index, the object is restored by querying its index from the local object map.

Parameter buffering. ADSENTRY enforces security policies on the result of JavaScript actions, which will be described in Section 3.3.4. One possible way to bypass our access control policy enforcement is to insert content into the web page piece by piece. For example, instead of calling

```
document.write("<scr" + "ipt> some script <"  
              + "/scr" + "ipt>");
```

malicious ad script may attempt to avoid being detected by inserting a `script` element like the following

```
document.write("<scr");  
document.write("ipt> some script <");  
document.write("/scr");  
document.write("ipt>");
```

This way, checks on parameters to each individual DOM function call would not detect that a new `script` element is being inserted. To prevent such misuses, ADSENTRY buffers such consecutive calls to `document.write` by not sending them one by one to the shadow JavaScript engine for execution, but finally replaces them with a single call with the entire piece of content being inserted, which is checked by the access control policy enforcer as normal. Although aggregating multiple program statements into one is a difficult problem in general, we find our technique quite effective for segmented calls to `document.write` among ads scripts.

3.3.4 Access Control Policy Enforcement

To regulate the communication between the host web page and the confined ad script, our policy enforcer acts as a moderator. Any communication between the two parties needs to be approved according to a given policy. ADSENTRY is flexible in allowing both web publishers and end users to specify the access control policies for ads.

ADSENTRY has a default policy. The default policy disallows any JavaScript code originated from ads to run in the host web page. In other words, all untrusted scripts will be guaranteed to be only executed inside the shadow JavaScript engine. To enforce

that, we examine all incoming messages from the sandbox, distinguish page updates containing dynamic JavaScript content versus static HTML, and then handle them accordingly.

Specifically, for the static HTML content, our system first normalizes the HTML into the corresponding XML format and then serializes the XML back to HTML before processing. The HTML code is widely known as badly formed, to the point that badly written code is often called “tag soup” [175]. Also, all major browsers have permissive parsing behaviors by supporting a rendering mode called “quirks mode” beside the “standards mode” [174]. These browser quirks have many negative implications, one of which is that malicious attacker can embed JavaScript code inside a malformed fragment of HTML code. To strive a balance between security and the support of potential browser quirks, we took three phases for parsing HTML code. First, we attempt to reformat the code by correcting popular mistakes in web authoring. For instance, we close all open tags and correct all improperly nested tags. Second, we leverage the XML parser in the web browser to parse this reformatted code into a XML model. Note that a malformed HTML is considered dangerous and will be rejected by our parser. Since XML parser is strictly standard-compliant, any surviving formation will bear no ambiguity. Finally, we serialize this XML model back to HTML code before handing to the page agent for further processing.

For the JavaScript dynamically generated by ads scripts, we install wrappers that request the sandbox to run the dynamic JavaScript code. In other words, all untrusted scripts are guaranteed to execute inside the shadow page, not the real page. In our prototype, we apply the code wrapping based on the above XML model. Specifically, we leverage the XML XPath facility available in most browsers to traverse the XML model tree and inspect the enclosed nodes. We first query for patterns of dynamic code on the model. These patterns of dynamic code include event handlers such as `onclick()`, as well as related JavaScript functions such as `addEventListener()`, `setTimeout()` and `setInterval()`. The resulting node set will then be properly wrapped or transformed. In our prototype, we have installed wrappers on all 32 possible vectors of dynamic code and ensure that no potentially malicious code will ever be injected to the real page. As an example, the following code snippet

```
setTimeout(' slideAd(10,100); ', slideDelay);
```

will be transformed into the following code fragment:

```
setTimeout(' sandboxAds(" slideAd(10,  
100);", id, false); ', slideDelay);
```

To further ensure the privacy of sensitive user data in the web page, we allow users to configure the data to be shared with the script. As mentioned earlier, we do not copy all the content of the real DOM to the virtual DOM. Instead, we choose to interpose on every access to the virtual DOM from the untrusted ad and subject it for policy verification. As such, users can decide to be extremely cautious with certain kind of ads, and block any read access from the ad to the entire page. On the other extreme, a user might want to trust certain ads, and allow free accesses to the real DOM content. In addition

Bugzilla ID	Attack Behavior	Outcome
426520	Browser crashed by memory corruption with crafted XML namespace	Contained by shadow JS engine
454704	Browser crashed by exploiting a vulnerability of XPCSafeJSObjectWrapper	Contained by shadow JS engine
465980	Browser crashed by pushing to an array of length exceeding limit	Contained by shadow JS engine
493281	Browser crashed by stack corruption starting at unknown symbol	Contained by shadow JS engine
503286	Browser crashed by exploiting a vulnerability of Escape()'s return value	Contained by shadow JS engine
507292	Browser crashed by incorrect upvar access on trace involving top-level scripts	Contained by shadow JS engine
561031	Browser crashed by overwriting jump offset	Contained by shadow JS engine
615657	Browser crashed by buffer overflow due to incorrect copying of upvarMap.vector	Contained by shadow JS engine

Table 3.1: ADSENTRY Evaluation Using Browser Exploits

to the above two policies, a user is also allowed to specify a policy that blocks accesses to the `document.cookie` object or mandates that ad can only read from its own elements and not the surrounding content. Moreover, an ad can be prohibited from appearing outside of the allocated region of the web page (by stating the allowed values of `width`, `height` and `overflow` property of ad elements). This is helpful to thwart some types of phishing attacks. In fact, as a comprehensive isolation framework, our system provides a mediation capability that can accommodate existing access control policies [158] for ads. And both web publishers and end users can take the advantage of the same capability to enforce security policy on ad behaviors.

Beyond the default policy ADSENTRY supports more complex access control policy to constrain the access from ads to the web page DOM, and to the network. We focus on the default policy and we test with detecting and preventing common privacy violation attempts in our evaluation. However, it is also possible to enforce more complex policies, such as ring-based access control [82]. ADSENTRY provides the infrastructure for enforcing different policies for untrusted ads, but devising and verifying effective access control policy would merit separate research, which is beyond the scope of this thesis.

3.4 Evaluation

In this section, we evaluate the functionality and performance of ADSENTRY. In particular, we have conducted four sets of experiments. The first one is based on real-world browser exploits to evaluate ADSENTRY's defense against drive-by download attacks. The second one is to test its resilience against malicious attempts that inject JavaScript into web applications. The third one is to evaluate ADSENTRY's protection of privacy against rogue information-stealing ads; The fourth one is to measure the performance overhead. Our experiments were conducted on a Dell E8400 workstation with a Core 2 Duo CPU (3GHz 6 MB L2 Cache) and 4GB of RAM. The system runs Ubuntu 9.10. We mainly use its default web browser – Mozilla Firefox 3.5.8 – for our experiments.

Scenario	Attack Vector	Attack Behavior	Outcome	Description
1	Direct code injection	Inject script	Blocked	Denied by the default policy
2	Browser parsing quirk	Malformed <code></code> tag	Blocked	Rejected by message normalization
3	Browser parsing quirk	Malformed <code><script></code> tag	Blocked	Rejected by message normalization
4	Browser parsing quirk	Malformed <code><script></code> tag	Blocked	Rejected by message normalization
5	Browser parsing quirk	Malformed <code></code> tag	Blocked	Rejected by message normalization
6	Browser parsing quirk	Malformed <code><script></code> tag	Blocked	Rejected by message normalization
7	Browser parsing quirk	Malformed <code><iframe></code> tag	Blocked	Rejected by message normalization

Table 3.2: ADSENTRY Evaluation Using JavaScript Injection Attacks

When evaluating on specific JavaScript engine exploits, we evaluate on the corresponding version of Firefox, such as Firefox 3.0.

3.4.1 Browser Exploits

To evaluate the effectiveness of ADSENTRY in sandboxing ads, we conducted experiments with a few real-world exploitations, obtained from existing research work [97] as well as vulnerability databases [108, 107]. All the exploits we tested with caused the vulnerable versions of the Firefox browser to crash during our experiments. They are all marked as critical by Mozilla developers, and can be further crafted to launch severe attacks such as drive-by download.

Our experiments are summarized in Table 3.1. The eight examples exploit the vulnerabilities in the SpiderMonkey JavaScript engine. Most of them are various instances of buffer overflow or memory corruption attacks, and they could lead to arbitrary code execution. With ADSENTRY installed in the vulnerable versions of the Firefox browser, each of the exploits was successfully contained by the shadow JavaScript engine. This confirmed and demonstrated one of our design goals that we would like to run untrusted ads scripts in an isolated environment so that even in the worst case, they would not crash the entire web browser. As ADSENTRY sandboxes the JavaScript engine, so any memory attack against vulnerabilities in the JavaScript engine would be contained by the sandbox.

3.4.2 Script Injection by Ads

In our second experiment, we evaluated the effectiveness of our default policy in preventing untrusted code from being injected from the ad to the web page. In particular, we examined the XSS Cheat Sheet [141] and identified a number of cases that can successfully result in injecting JavaScript from the ad into the web page for execution. We confirmed the successful injection and execution in the default Firefox without ADSENTRY being installed. During our experiments, we explicitly cleared the browser’s cache between each step.

Our results are shown in Table 3.2. The first one is a direct attempt to include an external JavaScript to execute in the web page while the other six exploit numerous parsing quirks [141]. Such attacks are created to execute a simple script that displays a message box “hacked!” The use of browser parsing quirks reflects the current trend [159] in part because they are much harder to repair without breaking compatibilities with legacy web applications. This was blocked by the default policy in ADSENTRY that direct injection of scripts into the web page is disallowed.

For the rest examples, we use the second scenario as the representative. Specifically, in the second scenario, the attempt is to exploit a parsing quirk by embedding a `<script>` tag as literal text inside a `` tag, which will cause the browser to interpret the text string as JavaScript code, thus causing an injection:

```
<IMG """><SCRIPT>alert ("XSS")</SCRIPT>>
```

The related code snippet is shown above. It contains three pairs of double-quotes, encapsulating different parts of the text. If a parser were properly implemented, there would be three literal strings: an empty string `" "`, the second string `"><SCRIPT>alert ("` and the last string `)</SCRIPT>"`. These three strings are orphaned as they are not assigned to any property of the tag and therefore should be discarded. As such, the entire tag should simply collapse to ``, which can also be disregarded. However, this is not the case in most modern browsers. In fact, existing browsers tend to be very permissive in their parsing behavior [174]. For instance, we observed that Firefox interpreted `` as the first tag and `<SCRIPT>alert ("XSS")</SCRIPT>` as the second tag; the remaining `">` was accepted as plain text and displayed as is. As a result, the “malicious” code `alert ("XSS")` was executed. This attempt was blocked because of the normalization through the standard-compliant XML in our system. We successfully detected this malformed HTML content and substituted it with the benign static text “Script Injection Blocked.”

3.4.3 Privacy Protection

In our third set of experiments, we test our system from the privacy perspective. In particular, it has been known that third-party JavaScript can violate user privacy in various ways. Examples include cookie stealing, location hijacking, history sniffing, and behavior tracking [81]. In our experiments, we evaluated ADSENTRY with a synthesized ad that simulates the above information-stealing behaviors.

In particular, the synthesized ad is developed to perform all these four types of behaviors: The cookie stealing is implemented to access the `cookie` property of `document` object; The location is hijacked by setting the `location` property of `window` (or `document`); The previously browsed URLs are sniffed by obtaining the color of the populated hyperlinks, which can be done by invoking the `getPropertyValue` function of the `ComputedCSSStyleDeclaration` object (with the argument “color”)²; Behavior tracking is achieved by registering related event listeners of interested ele-

²Recent browsers return the same computed styles for visited and unvisited links.

Website	Properties of Ads
www.msn.com	Ads on different domain of same company
www.aol.com	Ads on content distribution network (CDN)
www.livejournal.com	Ad network DoubleClick
espn.go.com	Ad network DoubleClick
www.cnet.com	Ads on different domain of same company
imageshark.us	Ad network Google
www.nytimes.com	Ad network Checkm8
www.ehow.com	Ad network YieldManager
sourceforge.net	Ad network DoubleClick
www.reference.com	Ad network DoubleClick
www.dailymail.co.uk	Ad network DoubleClick
www.guardian.co.uk	Ad network Google
www.gmx.net	Ad network Uimserv
yfrog.com	Ad network Rubicon Project
www.comcast.net	Ad network Yahoo!

Table 3.3: Websites Used in User Experience Evaluation

ments, such as `onclick`, `onmouseover`, etc.

ADSENTRY successfully detected each of the above four types of behaviors. For the first two types, our system simply denies the `document.cookie` and the write access to the `window.location` and `document.location`. For the third type of ad behavior, it is detected by monitoring any invocations to the related `getPropertyValue` function. For behavior tracking, ADSENTRY refused the registration of callback routines of those elements that we marked as non-readable by ads. We discuss more on the tool for specifying access permissions in the next paragraph. We stress that our privacy protection enforcement does not suffer from JavaScript object and property aliasing problems. This is because the access is intercepted by the virtual DOM that, when invoked, has already resolved all object and property aliasing, if any.

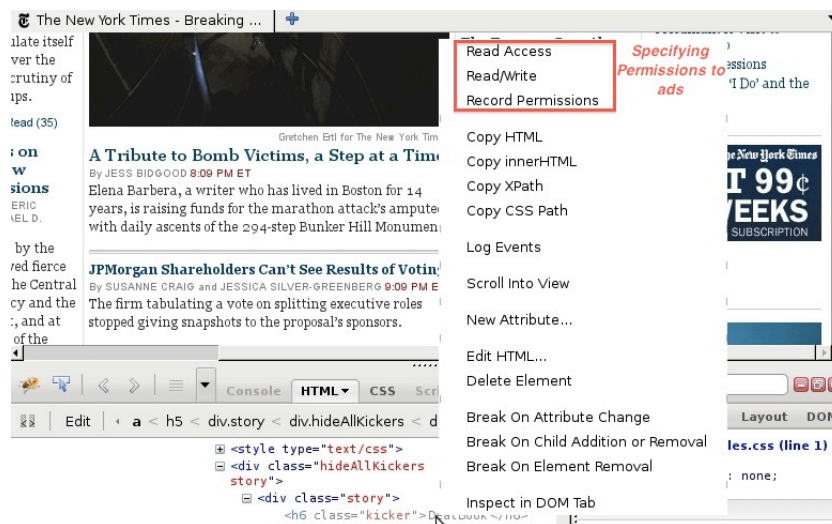


Figure 3.2: A Firebug-based Tool to Facilitate Specifying Access Permissions for Ads

We also evaluated the user experience of ADSENTRY using 15 popular website with ads, shown in Table 3.3. The embedded ads are automatically recognized by the Ad-block Plus extension and then transparently confined with ADSENTRY. To allow users to interactively specify security policies, we integrate a Firefox extension called Fire-

Performance Test	with ADSENTRY (<i>ms</i>)	without ADSENTRY (<i>ms</i>)	Overhead (%)
Google AdSense Rendering	381	363	4.96
DoubleClick Ad Rendering	601	578	3.98
MSN Ad Rendering	1224	1188	3.03
Yahoo Ad Rendering	1539	1475	4.34

Table 3.4: Runtime Page Load Overhead of ADSENTRY

bug [53] and extend it with a pop-up menu that can be triggered with a right mouse click (shown in Figure 3.2). Specifically, we use the Firebug to visually capture available screen regions and for a selected region, a right mouse click will activate the pop-up menu. From the menu, a user will be shown the list of ads (grouped by domains) currently embedded in the current page and can then choose which ad can have a read access to the chosen screen region or can register call-back routines (e.g., event listeners). By default, these ads are only allowed to read their own elements, not the surrounding areas. Users can also specify new policies during run time, which will overwrite existing ones if necessary.

Our experiments did not find any suspicious information-stealing behavior for these websites.

3.4.4 Performance Evaluation

In order to assess the performance overhead, we conducted experiments to measure the page load overhead. We picked four banner ads from the embedding websites, one for each of the top four ad networks. We created a test page for each ad and ran the test page with and without ADSENTRY. Each experiment was repeated for 20 times, and the average results were recorded.

Our results are shown in Table 3.4. Overall, ADSENTRY incurs small overhead. The relative overhead ranges from 3.03% in MSN Ad Network ad to 4.96% in Google AdSense ad. We observed that a typical ad might only infrequently access DOM namespace, which might attribute to the low overhead. From another perspective, the relative overhead can be low because ad content such as images are often dynamically loaded from a remote server, this process experiences network round trip delay that is typically much more significant than local computation time in web browsers. Also, to improve responsiveness, modern browsers typically start rendering any elements immediately once they are available. Therefore, a user may not notice the difference in the speed of ad loading time at all. In other words, this pipelining of the rendering process contributes to masking the delay that may be experienced by any single element in a web page.

To further evaluate the potential performance overhead, we apply 2 microbenchmarks. We measure the time needed to initialize our sandbox. Our results show that it takes 31 ms to initialize and set up the sandbox. Though it is lightweight, we expect opportunities still remain to reduce the time by further optimizing the JavaScript engine and NaCl sandbox. Finally, we evaluate a round-trip communication delay for a virtual DOM access. Without our system, it typically took 0.001 ms for the ad to finish

the reading of a particular DOM property. When being confined, it will take 0.59 ms. This is expected as it needs to cross the sandbox boundary and go through the normalization for policy verification. Note that this overhead will be effectively amortized in real-world scenarios – as demonstrated in the four real ads.

3.5 Discussion

In this section, we discuss the limitation of ADSENTRY and future work. First, our current work focuses on the JavaScript-based advertisements and has not yet explored the support of other types of advertisements. In particular, Flash technology is another popular way to write and display ads, which still remains to be investigated how flash-based ads can be supported.

Second, ADSENTRY protects the browsers from attacks exploiting vulnerabilities of the JavaScript engine, but it is not designed to prevent attacks to other browser components, such as the HTML rendering engine. If the malicious HTML segment is dynamically generated by JavaScript code, ADSENTRY’s policy engine can mitigate the attack by the HTML normalization and signature-based attack blocking. A more general solution is to extend our solution to isolate other components of the browser.

Our prototype implementation is able to handle typical JavaScript advertisements, which has limited ways in accessing other parts of the web page. However, third-party JavaScript code in general has much tighter integration with the rest of the web page. As our future work, we will improve the support for transparently isolating a wider class of JavaScript code in web applications. It will also be interesting to investigate possible ways (e.g., in software testing) that automatically test ADSENTRY’s compatibility with a broader set of web applications.

Lastly, it is possible for malicious ads to monopolize the execution, thus preventing other regular scripts from being executed. However, this issue also exists with web browsers today. Although ADSENTRY is not designed to prevent denial-of-service attacks, it can mitigate them by enforcing certain timeouts in the Policy Enforcer, during the execution of untrusted ads.

3.6 Related Work

In this section, we discuss existing work that mitigates threats from untrusted web content embedded into web applications, including those compromising user data, web application integrity as well as users’ operating systems.

Drive-by download prevention. Drive-by downloads are serious threats to web and host security [132, 133]. BLADE [97] proposes a detection system for drive-by download exploits. This type of attacks has recently received lots of attention. For example, heap-spraying attacks can pre-populate a large heap space with attack code and a software bug can be exploited to redirect execution flow to the heap sprays (with attack

code). In addition, several systems [49, 134, 42] have been proposed to leverage specific memory characteristics of these attacks to identify them and prevent browsers from being exploited. WebShield [92] proposes a middlebox framework that processes page contents in a shadow browser, and transforms DOM updates to the client browser to reflect DOM changes there. As a result, drive-by downloads can be detected at the middlebox without affecting the client browser. Other existing sandbox and isolation solutions [60, 93] can also be used to protect the operating system against drive-by download attacks. Compared to ADSENTRY, solutions in this category are not designed to protect user privacy and web application integrity from malicious JavaScript ads.

Isolation in web browsers. Several recent research projects [62, 36, 169] attempt to achieve better browser security architecture by running different browser components in isolated environments. The Google Chrome browser also uses a sandbox to isolate browser components and protect the operating system [104, 15]. The IBOS [154] system steps further by designing a secure architecture for both the operating system and the web browser altogether, minimizing default sharing and trust between software components. However, they do not support isolating JavaScript ads from the rest of web applications, while ADSENTRY executes untrusted ads scripts in a separate and sandboxed environment from trusted scripts, mediating every access from ads to web applications.

Web application integrity protection. To prevent tightly-integrated third-party JavaScript from affecting the integrity of web application, one type of solutions [38, 50, 64, 99, 98, 52] restricts the “dangerous” functionality of JavaScript. For example, ADsafe [38] only allows ads to use a safe subset of the JavaScript functionality. It removes dangerous JavaScript features, such as global variables, `eval`, `this`, and `with`. ADSafety [126] proposes a lightweight and efficient verification for JavaScript sandboxes, and has been successfully applied to ADsafe. Another line of solutions [156, 79, 125, 135, 180] protects web application against JavaScript ads through code transformation, enforcing policies against malicious JavaScript at runtime. Similarly, ConScript [105] introduces aspect into JavaScript language to enforce users’ security rules. MashupOS [168] proposes new script integration primitives reflecting different trust relationships between the integrator and the mashup content provider. Besides enabling web publishers to protect their web applications, ADSENTRY also allows end users to flexibly specify access control policies according to their own requirements.

AdJail [158] addresses the privacy and web application integrity threat from ads by isolating them into an iframe-based sandbox. Using a separate origin in the sandbox, AdJail leverages browser’s native origin-based protection to isolate ads. It is a solution for publishers to isolated third-party ads. Compared to ADSENTRY, AdJail assumes the ads on a web page are relatively independent and do not have tight dependencies with the page environment. For example, ad scripts cannot access global JavaScript objects defined or overwritten by other trusted scripts in the same hosting page. ADSENTRY

transparently supports tight dependency between ads and the host page, without significant modification of the web page. It also provides flexible control of behaviors of JavaScript ads.

In addition, solutions in this category cannot prevent malicious ads from exploiting browser vulnerabilities.

Privacy protection. One of users' major concerns about JavaScript ads is privacy. Privad [66] proposes a solution to protect users' privacy by making users anonymous to the advertisers and publishers, but it does not prevent users' data from being used by the ad script, which may implicitly leak our user data. Adnostic [160] uses a browser extension to perform ad targeting, selecting ads to display from a larger set of ads sent by the advertisement network. Compared to ADSENTRY, both solutions only focus on protecting users' privacy, and do not address the ad's threat to integrity of web applications and the underlying operating system.

3.7 Summary

JavaScript-based advertisements are ubiquitous on the Internet. They pose threats to the privacy and integrity of web applications, as well as security of operating systems. In this work, we present the design, implementation, and evaluation of ADSENTRY, a comprehensive and flexible framework to confine untrusted JavaScript advertisements. ADSENTRY not only separates the untrusted ad execution in a shadow JavaScript engine, but also mediates their access to the main page with access control policies, which can be specified by both web publishers and end users. We have implemented a Linux-based prototype of ADSENTRY that supports current Firefox browsers. Our experiments with a number of ad-related exploits show that ADSENTRY is effective in blocking these attacks. Our performance evaluation shows that the comprehensive protection is achieved with a small performance overhead.

Chapter 4

A Quantitative Evaluation of Privilege Separation in Web Browser Designs

Privilege separation is a fundamental concept for designing secure systems. It was first proposed by Saltzer et al. [142] and has been widely used in redesigning a large number of security-critical applications [131, 18, 15]. In contrast to a monolithic design, where a single flaw can expose all critical resources of a privileged authority, a privilege-separated design groups the components of a system into *partitions* isolated from each other. The principle of least privilege suggests that each partition must be assigned the minimum privileges it needs for its operation at run-time. Intuitively, this reduces the risk of compromising the whole system. In a privilege-separated design, by exploiting a vulnerability the attacker gains access only to the small subset of privileges granted to the vulnerable component.

Web browsers are the underlying execution platform shared between web applications. Given their importance in protecting against threats from the web, web browsers have been a prime area where privilege separation is being applied. For instance, numerous clean-slate browser proposals [15, 36, 169, 62, 63, 75, 154, 92] and commercial browsers like Bromium [1] and Invincea [2] are customizing privilege separation boundaries in web browsers. In retrofitting the principle of least privileges to web browsers, numerous important design questions are actively being debated. Should browsers put each web origin in its own partition? Should browsers host sub-resources (such as images, SVG, PDF, iframes) of a web page in separate partitions? Should sub-resources belonging to one origin be clubbed into the same partition? Should two code units (say, the JavaScript engine and the Document Object Model (DOM)) be assigned to different partitions? A systematic methodology to seek answers to these questions is important, but has not been investigated so far.

In this work, we argue that these design decisions are not completely governed by security demands. Security benefits of privilege separation can come at the cost of *performance*. If we isolate browser components too much, the cost of communication

between these components can pose a heavy performance penalty. Then, significant development effort must be invested to overcome the performance bottlenecks that can lead to increased investment in development costs. A practical privilege-separated design should balance security with other competing concerns such as performance.

Problem & approach. How does one estimate the performance bottlenecks that will arise after redesigning a web browser with a chosen privilege partitioning configuration? In this work, we take a first step towards quantitatively studying this question. We seek a measurement-based methodology to identify the security benefits and performance costs incurred by a privilege-separated design. This methodology is based on widely-used practices in system designs. Security architects often use similar back-of-the-envelope calculations to identify these bottlenecks before a design is finalized. However, such reasoning has been based on the security architect’s intuition and domain knowledge, and has not been scientifically studied in prior research. In previous research on privilege-separated browsers, performance measurements have been “after-the-fact”, *i.e.*, after a chosen partitioning configuration has been implemented, and on a small scale (typically on 5-10 sites). We suggest evaluating these performance costs and benefits upfront on a larger-scale dataset.

Our methodology aims to measure weak upper bounds on the performance incurred by a proposed browser partitioning scheme. These bounds can, of course, be reduced in real implementations with careful use of communication channels and/or redesign of the interfaces between components. However, this methodology lets us identify the likely bottlenecks where significant engineering effort needs to be invested. On the security front, most prior works (somewhat informally) argue security based on two artifacts: (a) the reduction in size of the trusted computing base (TCB), and (b) the reduction in number of known vulnerabilities affecting the TCB after the redesign. To unify the security arguments previously proposed, we systematically measure these using real-world large datasets. Specifically, our methodology takes a conceptual blueprint of the web browser components, an annotated vulnerability database, and the application executable as the input, and measures parameters over a large test harness. The empirical data from the measurements are then leveraged to quantify security benefits and performance costs in design dimensions of a given privilege-separated design. Based on such empirical quantification, security architects can make trade-offs between security and performance, compare two partitioning configurations, and iterate quickly over the proposed partitioning design before implementation.

We develop an assistance tool PRIVGAUGE to automate our measurements to a large extent. We apply PRIVGAUGE to study the design decisions in 9 recent browser designs that are being debated actively. Our approach successfully scales to Mozilla Firefox, which is a complex web browser with 8 years of development history and over 3 million lines of code.

Browser	Isolation Primitive	Partitioning Dimension	Plugins	JS	HTML Parser	DOM	Layout	Network	Storage
Firefox	Process	Nil	Separate	⊕	⊕	⊕	⊕	⊕	⊕
Chrome	Process	By Origin, By Component	With Hosting Page or Separate	⊕	⊕	⊕	⊕	○	○
Tahoma	VMs	By Origin	With Hosting Page	⊕	⊕	⊕	⊕	⊕	⊕
Gazelle	Process	By Origin, By Sub-resource, By Component	Separate Per Origin	⊕	⊕	⊕	⊕	○	○
OP	Process	By Origin, By Component	Separate Per Origin & Plugin	⊕	○	○	○⊗	◇	⊖
OP2	Process	By Origin, By Sub-resource, By Component	Separate Per Origin	⊕	⊕	⊕	⊕	◇	⊖
IE8/9	Process	Per Tab	With Hosting Page (ActiveX)	⊕	⊕	⊕	⊕	⊕	⊕
IBOS	Process	By Origin, By Sub-resource, By Component	Separate	⊕	⊕	⊕	⊕	○	⊗
WebShield	Host	Nil	With Hosting Page	⊕	⊕	○	○	⊕○	⊕

Table 4.1: Privilege Separation in Browsers *The table explains different partitioning dimensions in browser designs. For the right part of the table, same symbols denote the corresponding components are in the same partition. e.g., in Chrome, JS, HTML Parser, DOM, and Layout belong to one partition (⊕), while Network and Storage belong to another partition(○).*

Our results. We provide the first detailed measurement of the security benefits and performance costs in the recent 9 browser designs. Our measurements lend pragmatic insights into some of the crucial design questions on how to partition web browsers. For example, isolating web origins incurs negligible overhead in client-side cross-origin communication, whereas using separate processes to load cross-origin sub-resources would on average require 51 processes per web site.

The results of our work correlate well with browser redesign proposals that have been adopted. We also identify the key performance bottlenecks in our browser blueprint. For instance, isolation between the JavaScript engine and DOM would create a performance bottleneck. Detailed results are presented in Section 4.4. We hope our browser blueprint and identified bottlenecks will be instructive to understand how existing commercial browsers and future designs overcome these practical constraints we observe.

Our goal in this work is not to suggest new browser designs, or to undermine the importance of clean-slate designs and measurement methodologies proposed in prior work. On the contrary, without extensive prior work in applying privilege separation of real systems, the questions we ask in the work would not be relevant. However, we argue to unify the methodology in quantifying the trade-offs of a privilege-separated design and to reason about it on a more systematic foundation. Although our explanation of the methodology and the foundations is on web browsers, our main arguments can be more generally applied in privilege separation of other applications too.

4.1 Overview

In this section, we introduce the concept of privilege separation, and then discuss privilege-separated designs in web browsers, including their goals and various design dimensions.

4.1.1 Privilege Separation in Concept

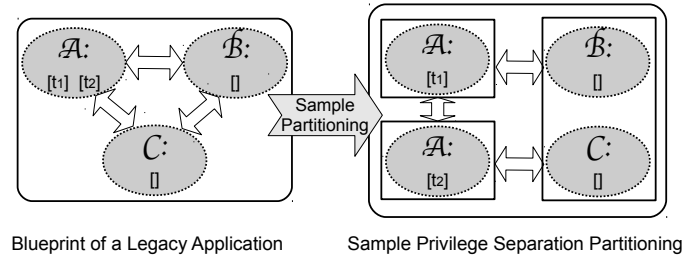


Figure 4.1: Overview of Privilege Separation

Web browsers, just like operating systems, aim to securely manage access to the resources of different *authorities* such as web origins. The executing instruction is part of a *code unit*, which can be defined to be a logical component, a method, a program statement, or even a block of instructions. Privilege separation divides resources of different authorities into different partitions, and limits code units to access resources outside their partitions. Figure 4.1 illustrates a blueprint of an application’s code units and run-time *authorities* (in brackets) each code unit may carry at any given instant during its execution. In Figure 4.1, the code unit \mathcal{A} has two instances running with separate authorities, t_1 and t_2 , whereas code units \mathcal{B} and \mathcal{C} are shared resources such as libraries, which do not carry any default authorities.

Privilege separation aims to determine how to minimize the attacker’s chances of obtaining unintended access to another authority’s resources. Specifically, let p_i be the probability for any authority other than i to get unintended access to resources r_i belonging to i . From a purely security perspective, the goal is to minimize the attacker’s advantage. We can model this advantage using a variety of mathematical functions. For instance, an attacker’s worst-case advantage from compromising a single vulnerability may be defined as $\max(p_i)$; a privilege separated design is good if it yields a large quantity $1 - \max(p_i)$ ¹. However, as we argue in this work, a practical privilege-separated design often departs significantly from this conceptual formulation. We argue that this purely security-focused viewpoint ignores the implicit *performance* costs associated with partitioning. Rather than focusing on mathematical modeling, we focus on the key methodology to quantify the benefits of a privilege partitioning scheme in this work.

¹Alternative definitions of attacker’s advantage are easy to consider—for example, considering the average case with *avg* rather than *max*. We can assign additional weights to the resources r_i via a severity function $\mathcal{S}(j, r_i)$ if failing to protect r_i from j has higher severity than other resources, etc.

4.1.2 Privilege Separation in Browsers

Blueprint. To discuss trade-offs in partitioning, we use a conceptual blueprint that shows the various code units in a typical browser. We have manually extracted this from Mozilla Firefox, a popular web browser, as shown in Figure 2.1 in Section 2². We have confirmed that this conceptual blueprint is also consistent with WebKit-based browsers and models sufficient details for comparing prior works on browser redesign. This blueprint intuitively explains the processing of web pages by various browser components, which are the code units we consider in this study. A web page is first received by the Network module that prepares content to be parsed by the HTML parser. The HTML parser creates a DOM, which can then invoke other execution engines such as the JavaScript engine, CSS, and so on. The legitimate flow of processed content between components is illustrated by time-ordered arrows; for brevity, we skip explaining the details. In a single-process browser, all these components execute in the same partition. Web browser designs utilize privilege separation to isolate the resources owned by different authorities, which are defined next.

Isolating authorities. Web browsers abstractly manage resources owned by one of the following authorities:

- *Web origins.* A web origin is defined by the same origin policy as the port, protocol and domain of the web site serving web content. All content belonging to a web origin including HTML data, sourced scripts, CSS and so on in a web session is owned by the active web origin [4].
- *System authority.* This authority denotes the core part of the browser code and storage, which is also sometimes referred to as the chrome privilege. A number of sensitive resources belonging to the underlying OS system, such as the file system, network, display, and so on, are owned by the system authority.
- *User authority.* Certain resources are granted discretionary access based on the consent by human users [139]. Several UI elements in the browser are expected to convey to users the correct and necessary security indicators to allow them to make sensible security decisions, such as security prompts, certificate warnings, access to preferences and settings. These resources are conceptually grouped to belong to the user authority.

Security threats. Security vulnerabilities can result in one authority gaining unintended access to resources of another. In web browsers, we can classify threats based on which authority gains privileges of which other authority.

- *CROSS-ORIGIN: Cross-Origin Data & Privilege Leakage.* Vulnerabilities that allow access across origins are considered under this category of vulnerabilities.

²Security analysts can pick different blueprints in their design; our methodology is largely agnostic to the blueprint used.

Bugs in enforcing SOP fall into this category, such as missing checks on a certain path of cross-origin access, capability leaks [17], and leaking descriptive information such as XMLHttpRequest status or heap addresses.

- *WEB-TO-SYS: Web-to-System Privilege Escalation.* A strictly more severe category of vulnerabilities permits a web origin the full privileges of the system (or chrome) privileges. This category includes vulnerabilities that allow access to system authority resources via limited interfaces (such as exposed JavaScript APIs), but do not necessarily grant arbitrary code execution capabilities (next category). Vulnerabilities in browser plugins, browser helper objects, and components that run with default privileges of the system authority lead to these bugs.
- *WEB-TO-COMP: Web-to-Component Privilege Escalation.* This category includes the vulnerabilities that could potentially allow attackers to run arbitrary code in the privilege of a vulnerable browser component. Vulnerable browser components may or may not run with system privileges at the time of exploit. This category includes memory corruption vulnerabilities that allow arbitrary code execution in the component’s runtime authority.

There are also other categories of browser vulnerabilities. For completeness, we list them below. However, these are beyond the scope of the same-origin policy and we do not measure the security benefits of applying privilege separation to mitigate them.

- *USER: Confusion of User Authority.* These vulnerabilities may allow attackers to manipulate user interfaces to confuse, annoy, or trick users, hijacking their abilities in making reasonable security decisions. Recent incidents of mistakenly accepting bogus or compromised certificates [171] also belong to this category.
- *INTRA-ORIGIN: Intra-Web-Origin Data & Privilege Leakage.* This category of browser vulnerabilities result in running code within the authority of a web origin. These include bugs in parsing malformed HTML content, identifying charsets, providing HTTP semantics and so on. They can introduce popular forms of web attacks, such as XSS, CSRF and so on.

Partitioning dimensions. We study the various dimensions along which 9 recent browser designs propose partitioning authorities to mitigate the aforementioned threats. Table 4.1 summarizes the design dimensions considered in each browser design, and we explain these dimensions below.

- *By origin:* Each origin has a separate partition. This mitigates CROSS-ORIGIN vulnerabilities between web pages. For example, IBOS [154], Gazelle [169], Google Chrome [15], OP [62] and OP2 [63] all isolate primarily on origins ³. In

³OP and OP2 propose isolating web pages within the same origin, but the same-origin policy does not recognize such intra-origin boundaries and permits arbitrary access between web pages of the same origin. From a security analysis perspective, we treat them as the same.

Chrome, by default web pages from different origins but belonging to the same “site instance”⁴ are an exception to this isolation rule.

- *By sub-resource*: When an origin is loaded as a sub-resource in another origin, say as an `iframe` or as an image, web browsers can isolate the sub-resources. This provides additional isolation between cross-origin resources, especially in mashups that integrate contents from various origins, and prevents CROSS-ORIGIN vulnerabilities from sub-sources explicitly included by an origin. For example, Gazelle [169] allocates a separate process for each destination origin of the resource and IBOS [154] uses a separate process for each unique pair of requester-destination origins; Google Chrome does not isolate sub-resources.
- *By component*: Different components are isolated in different partitions. Web browsers have proposed isolating individual components that are inadvertently exposed across origins, but not need the full privileges of the system authority. For example, the OP browser [62] isolates the HTML parser and the JavaScript engine in different partitions. This prevents an exploits of a WEB-TO-COMP vulnerability. Browsers also isolate components that need heavy access to resources of the system authority (such as the file system, network) from components that need only access to web origin resources. For example, Google Chrome [15] and Gazelle [169] separate components into web components (renderers) and system components (browser kernels). Partitioning along this dimension prevents WEB-TO-SYS vulnerabilities in the codebase of renderer partitions.

In this work, we apply systematic methodology to study the security and performance trade-offs in these partitioning dimensions.

4.2 Quantifying Trade-offs with Empirical Measurements

How do we systematically evaluate the performance and security trade-offs of a given partitioning configuration? To answer this question, we measure several security and performance parameters. Our methodology places arguments made previously on a more systematic foundation backed by empirical data.

4.2.1 Security Parameters

The goal of measuring security improvements is to estimate the reduction in the likelihood of an attacker obtaining access to certain privileged resources, which we introduced as probabilities p_i in Section 4.1.1. Estimating resilience of software to future or unforeseen has been an open problem [137, 94, 71]. In this work, our goal is not to investigate new metrics or compare with existing ones; instead, we aim to systematize measurements of metrics that have already been proposed in works on privilege

⁴Connected web pages from the domains and subdomains with the same scheme.[35]

separation. Security analysts argue improvements in security using two metrics: (a) reduction in TCB, i.e., the size of code that needs to be trusted to protect resource r_i , and (b) reduction in impact of previously known security vulnerabilities⁵. We explain the intuitive rationale behind these and systematically measure them using the following parameters S1-S3.

S1: Known vulnerabilities in code units. One intuitive argument is that if component A has more vulnerabilities historically than B, then A is less secure than B. Therefore, for a given partitioning scheme, we can compute the total number of vulnerabilities for code units in one partition as the vulnerability of that partition. The smaller the count, the less is the possibility of exploiting that partition to gain unintended access to its resources.

S2: Severity weightage. It is important to characterize the impact or severity of vulnerabilities. As we discuss in Section 4.1.2, different vulnerabilities give access to different resources. For instance, WEB-TO-SYS vulnerabilities give web attackers full access to system resources (including all other origins), so they are strictly more severe than CROSS-ORIGIN vulnerability. To measure this, we categorize security vulnerabilities according to their severity.

S3: TCB reduction. An intuitive argument is that if the code size of a trusted partition is small, it is more amenable to rigorous formal analysis and security analysis by human experts. If a resource r_i , such as the raw network access, is granted legitimate access to one component, then the size of the partition containing that component is the attack surface for accessing r_i . In security arguments, this partition is called the trusted computing base (TCB). By measuring the total code size of each partition, we can measure the relative complexity of various partitions and compute the size of TCB for different resources⁶.

We explain how we measure each of these metrics in Section 4.4.

4.2.2 Performance Parameters

The precise performance costs of a privilege-separated design configuration can truly be determined only after it has been implemented, because various optimization techniques can be used to eliminate or mitigate performance bottlenecks. However, implementing large redesigns has a substantial financial cost in practice. We propose a systematic methodology to calculate upper bounds on the performance costs of implementing a given partitioning configuration. These bounds are weak because they are calculated assuming a straightforward implementation strategy of isolating code units in separate

⁵Note that these metrics are instances of *reactive* security measurement, which have been debated to have both advantages [16] and disadvantages [137].

⁶We do not argue whether code size is the right metric as compared to its alternatives [29, 103]; of course, these alternatives can be considered in the future. We merely point out that it has been widely used in previous systems design practice and in prior research on privilege separation.

containers (OS processes or VMs), tunneling all communications over inter-process calls as proposed in numerous previous works on browser redesign. This strategy does not discuss any optimization that can be used in the final implementation. We argue that such a baseline is still useful and worthy of systematic investigation. For instance, it lets the security analyst identify parts of the complex system that are going to be obvious performance bottlenecks. Our methodology is fairly intuitive and, in fact, often utilized by security architects in back-of-the-envelope calculations to estimate bottlenecks. We explain the performance cost parameters **C1-C7** we are able to quantitatively measure below. Mechanisms for measuring these parameters and the inference from combining them are discussed in Section 4.4.

C1: Number of calls between code units. If two code units are placed in separate partitions, calls between them need to be tunneled over inter-partition communication channels such as UNIX domain sockets, pipes, or network sockets. Depending on the number of such calls, the cost of communication at runtime can be prohibitive in a naive design. If a partitioning configuration places tightly coupled components in separate partitions, the performance penalty can be high. To estimate such bottlenecks, we measure the number of calls between all code units and between authorities when the web browser executes the full test harness.

C2: Size of data exchanged between code units. If two code units are placed in separate partitions, read/write operations to data shared between them need to be mirrored into each partition. If the size of such data read or written is large, it may create a performance bottleneck. Two common optimizations can be used to reduce these bottlenecks: (a) using shared memory or (b) redesigning the logic to minimize data sharing. Shared memory does not incur performance overhead, but has trades-off security to an extent. First, as multiple parties may write to the shared memory regions, it is subject to the time-of-check-to-time-of-use (TOCTTOU) attack [122]; second, complex data structures with deep levels of pointers are easily (sometimes carelessly) shared across partitions that makes sanitization of shared data error-prone and difficult to implement correctly. To estimate the size of inter-partition data exchange, we measure the size of data that are exchanged between different code units. This measurement identifies partition boundaries with light data exchange, where Unix domain sockets or pipes are applicable, as well as boundaries with heavy data exchange where performance bottlenecks need to be resolved with careful engineering.

C3: Number of cross-origin calls. Client-side web applications can make cross-origin calls, such as `postMessage`, and via cross-window object properties, such as `window.location`, `window.top`, and functions `location.replace`, `window.close()`, and so on. We measure the number of such calls to estimate the volume of inter-partition calls if different origins are separated into different partitions.

C4: Size of data exchanged in cross-origin calls. Similar to **C2**, we also measure the size of data exchanged between origins to estimate the size of memory that may need to be mirrored in origin-based isolation.

C5: Number & size of cross-origin network sub-resources. One web origin can load sub-resources from other origins via network interfaces. If the requester is separated in a different partition than the resource loader, inter-partition calls will occur. We measure the number and size of sub-resources loading to evaluate the number of partitions and size of memory required for cross-origin sub-resource isolation.

C6: Cost of an inter-partition call under different isolation primitives. Partitioning the web browser into more than one container requires using different isolation primitives, such as processes and VMs. These mechanisms have different performance implications when they are applied to privilege separation. We measure the inter-partition communication costs of 3 isolation primitives in this work: Linux OS processes, LAN network hosts, and VMs; other primitives such as software-based isolation (heap isolation [13], SFI [167]) and hardware-based methods (using segmentation) can be calculated similarly.

C7: Size of memory consumption for a partition under different isolation primitives. With different isolation primitives, memory overhead differs when we create additional partitions in privilege separation. This is also an important aspect of performance costs dependent on design choices.

Measurement methodology. To measure the outlined parameters for a full web browser is challenging. We address this challenge by developing an assistance tool PRIVGAUGE that takes the following inputs:

- An executable *binary* of a web browser with debug information,
- A blueprint of the browser, including a set of code units and authorities for partitioning,
- A large test harness under which the web browser is subject to dynamic analysis.

We focus our measurements on the main browser components and we presently exclude measurements on browser add-ons and plugins. Our measurements are computed from data measured during the execution of the test harness dynamically, since computing these counts precisely using static analysis is difficult and does not account for runtime frequencies. With empirical measurements of the parameters we elaborate later, PRIVGAUGE outputs a database of empirical data that can assist security architects in finding privilege partition configurations with desired security and performance characteristics.

PRIVGAUGE is a binary-based measurement tool, and is applicable to different legacy applications. In our work, we apply it to a debug build of Firefox, a blueprint

manually abstracted from Firefox and WebKit designs, and historical Firefox vulnerabilities retrieved from Mozilla Security Advisories [55], and Alexa Top 100 websites (around August 2011).

4.3 Implementation of PRIVGAUGE

To apply our methodology to perform a large-scale evaluation, we develop a tool PRIVGAUGE to assist our measurements on the number of inter-component function calls and the size of data exchange.

The tool is built as a `pin` tool [77] for Linux with 1,600 SLOC. It instruments the Firefox binary executable on function invocations, returns, and instructions that have memory accesses. We conduct our measurement on a Ubuntu 10.04 64bit machine.

Re-establishing caller-callee information. For function call counting, PRIVGAUGE drives its execution of Firefox over the test harness of Alexa Top 100 websites. It dynamically instruments all function calls before they are invoked and before they return, respectively. For each running thread, it maintains a separate simulated call stack to record currently active functions. When a function is invoked, it pushes an entry with information detailed below onto the simulated call stack. Before a function returns, PRIVGAUGE pops out its corresponding entry on the simulated call stack for the running thread. This way PRIVGAUGE rebuilds the caller-callee information.

The entry generated for each function call contains a list of properties of the caller and the callee, including the original function name, the unmangled function name, the image name, the source file name, the source file line number, and the authority (if any). PRIVGAUGE extracts such information with interfaces provided by `pin`, with binaries built with debug information.

In the `pin` tool, we use `RTN_Insert` to insert our instrumentation function before and after the execution of each routine (function). Within the instrumentation function, we leverage `RTN_Name` to obtain the name of the function, and `PIN_UndecorateSymbolName` to remove C++ mangling. We call `RTN_Address` to get the starting address of the function, and `PIN_GetSourceLocation` to obtain relevant information of the source code file and line number of the current function in that file.

To maintain a separate simulated stack for each thread, PRIVGAUGE tracks the current thread ID with the `pin` parameter `IARG_THREAD_ID`, and uses `PIN_Mutex` to ensure synchronized access to PRIVGAUGE's own data from different threads.

Counting inter-partition function calls. PRIVGAUGE counts inter-partition function calls with the simulated stack. When a function call is intercepted by PRIVGAUGE, it retrieves its caller from the top of the simulated stack. PRIVGAUGE counts the number of function calls whose caller and callee belong to each pair of different code units. These numbers will later be aggregated for each pair of different partitions to calculate

the numbers of run-time inter-partition communications, for a given partition configuration.

Measuring the size of data exchange. To measure the size of data exchanged between components, PRIVGAUGE instruments all instructions that have memory read or write access. When a memory region is written by an instruction, it associates a *tag* with that memory region. A tag corresponds to the currently executing code unit and its associated authority. When a memory region is read by an instruction, it checks whether the code unit and authority of the tag associated with memory region is different from those of the instruction. If so, it records the size of the memory region together with the writer’s and reader’s code units and authorities. For multiple read operations by the same code unit and authority to the same memory region of the same writer, PRIVGAUGE only records the first of such accesses.

In our implementation, we use `INS_InsertCall` to instrument every instruction before and after execution. Within the instrumentation function, we call `INS_IsMemoryRead` to check whether the instruction is a *read* operation, and `INS_IsMemoryWrite` to check whether it is a *write* operation. Our tool also access parameters such as `IARG_MEMORYWRITE_EA`, `IARG_MEMORYWRITE_SIZE`, `IARG_MEMORYREAD_EA`, `IARG_MEMORYREAD_SIZE` for the address and size of memory region being written to and read from, respectively.

4.3.1 Challenge: Handling “Bridge” Components

In legacy applications, there may exist “bridge” components in the executable that do not appear in the blueprint of the legacy application. The main responsibilities of these components are just to bridge function calls between other components, such as the XPCOM component in Firefox. For our purpose of measuring inter-partition calls and data exchanges, communications with the bridge components are irrelevant. Therefore, PRIVGAUGE processes calls from and into them transparently, by only recording the actual callers and callees whose communications are bridged by any bridging component.

When measuring function call numbers across code units, PRIVGAUGE remembers the original caller for each call into the bridging components, and passes the caller information along the call sequences within the bridging components. Afterwards, when the function call goes out of the bridging components and the callee is another component, PRIVGAUGE only records a call from the original caller code unit to the first callee code unit out of the bridging components.

Similarly, when measuring data exchanges between code units, PRIVGAUGE always marks the writer of a memory region with the original caller code unit if the actual writer is a bridge component.

Category	Number of Vulnerabilities	
USER	20	
INTRA-ORIGIN	14	
CROSS-ORIGIN	38	
WEB-TO-SYS	11	
WEB-TO-COMP	277	JS:88, DOM:59, Layout:43, GFX:22, Plug-ins:12 Modules:10, XPConnect:10, Network:7, Security:5, Internationalization:4 Widget:4, Editor:3, Accessible:2, XPCOM:2, DocShell:1 General:1, Media:1, NSPR:1, Toolkit:1, URILoader:1

Table 4.2: Number of Historical Security Vulnerabilities in Firefox, Categorized by Severity and Firefox Components

4.4 Experimental Evaluation

To measure security benefits, we analyze the database of Firefox security vulnerabilities [55]. For the size of source code in different browser components, we use the `wc` tool to count the lines of source code for Firefox 8.0.

We measure performance overhead by running `PRIVGAUGE` over Alexa Top 100 websites [7]. Our study on these 100 websites gives us a comprehensive data set of all the run-time cross-code-unit calls and data exchanges.

We ran measurements on 3 isolation primitives: Unix domain sockets, network communications, and cross-VM communications over 10,000 times, with message lengths varying from 50 to 8K bytes. We measured memory overhead with processes, and VirtualBox and QEMU VMs.

Our measurements are mainly conducted on a Linux DellTM server with 2 Xeon[®] 4-core E5640 2.67GHz CPUs and 48GB RAM. For the measurement of inter-partition communication overhead, we connected two DellTM desktop machines with a dual-core i5-650 3.2GHz CPU and 4GB RAM via a 100 Mbps link.

4.4.1 Measurement Goals

Our measurements aim to figure out the following:

Goal 1. Security benefits of isolating a browser component with regard to the number of historical security vulnerabilities that can be mitigated.

Goal 2. Worst-case estimation of additional inter-partition calls and data exchange that would be incurred by isolating a component, and by isolating an authority (web origin).

Goal 3. Memory and communication overhead incurred by different isolation primitives.

4.4.2 Measurement over Alexa Top 100 Websites

We conduct our experiments on Alexa Top 100 websites, and obtain the following measurement results. Next, we explain how we measure these metrics and present their results.

Comp#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
LOC	136	367	74	155	32	3	131	21	77	366	10	269	763	17	223	24	137	478	24	188	53

Table 4.3: Kilo-lines of Source Code in Firefox Components. *In our experiments, we consider the following components: 0. NETWORK, 1. JS, 2. PARSER, 3. DOM, 4. BROWSER, 5. CHROME, 6. DB, 7. DOCSHELL, 8. EDITOR, 9. LAYOUT, 10. MEMORY, 11. MODULES, 12. SECURITY, 13. STORAGE, 14. TOOLKIT, 15. URILOADER, 16. WIDGET, 17. GFX, 18. SPELLCHECKER, 19. NSPR, 20. XPCONNECT, and 21. OTHERS.*

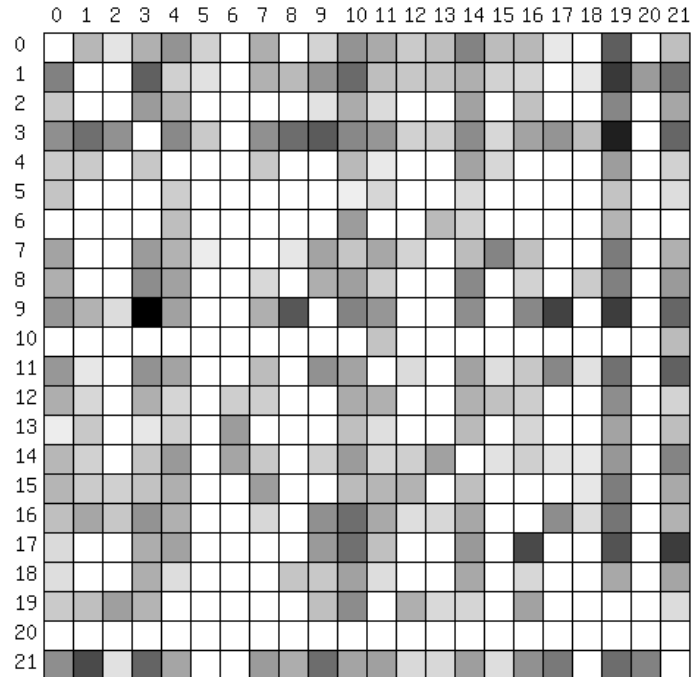


Figure 4.2: Gray-scaled Chart of Call Counts across Code Units. *Components are numbered as with Table 4.3. Each cell at (i, j) corresponds to the call counts between Code Unit i and j .*

For Goal 1: security benefits. We perform two measurements for measuring security benefits.

M1: *Number of historical security vulnerabilities in each Firefox component, categorized by severity.*

M2: *Size of source code in Firefox components.*

We measure historical Firefox vulnerabilities (Security Parameters **S1**, **S2**) with **M1**, and code size of different Firefox components (Security Parameter **S3**) with **M2**.

Measuring M1. We implement a Perl utility with 95 lines of code to crawl and fetch Firefox bug reports online [55]. According to the blueprint of browser components, and our classification of vulnerability severity, we count the 362 vulnerabilities we have access to⁷, by 1) *browser component*, and 2) *severity category*.

⁷2 of them are uncategorized due to insufficient information available.

Results of M1. Table 4.2 depicts the number of Firefox vulnerabilities with our categorization outlined in Section 4.1.2. We can see that 76.5% of the security vulnerabilities are WEB-TO-COMP vulnerabilities (277), which can lead to code execution. There is also a large amount of CROSS-ORIGIN vulnerabilities (38), whereas the number of other categories is much smaller. Among browser components, the JavaScript engine has the largest number of vulnerabilities (88). The Layout module (43) and DOM (59) also have large amount of vulnerabilities. These are all major components consisting of complex browser logic. On the other hand, more peripheral components, such as Editor has only 3 WEB-TO-COMP vulnerabilities. Such results are in line with our intuition that more complex and critical components tend to have more vulnerabilities discovered.

Measuring M2. We use the `wc` utility to obtain the number of lines of source code for all `.h`, `.c` and `.cpp` files in Firefox components.

Results of M2. Table 4.3 lists the number of lines of source code we measure for different components in Firefox. Components such as JavaScript, Layout and Security, etc. have large code size. These data reflect the (relative) complexity of different browser components (See **S3**).

For Goal 2: performance costs. We dynamically measure performance costs over Alexa Top 100 websites with the following. These measurements correspond to Performance Parameters **C1-C5**, respectively.

M3: *Number of function calls between browser components.*

M4: *Size of data exchanged between browser components.*

M5: *Number of client-side cross-origin calls.*

M6: *Size of data exchanged in client-side cross-origin calls.*

M7: *Number of different origins of sub-resources.*

Measuring M3 & M4. As described in Section 4.2, we apply PRIVGAUGE to browse Alexa Top 100 websites, counting the number of function calls whose caller and callee belong to two components, and the size of data exchanged during the process.

Results of M3. Table 4.4 lists the number of function calls (in 1000s) where the caller is in a different component than the callee when we browse over the Alexa Top 100 websites. These calls may become inter-partition calls after privilege separation. Thus, the larger the number is between the two components, the higher is the communication cost if they are isolated into different partitions. From the table, we see that there are 4,270,599,380 times of calls between the Layout engine and the DOM during our measurements, 369,305,460 times between the GFX rendering engine and the Layout engine, and 133,374,520 times between the JavaScript engine and the DOM. Heavy calls between these components correspond to tight interactions during run time, such as DOM scripting and sending layout data for rendering.

	NET- WORK	JS	PAR- SER	DOM	BROW- SER	CH- FOME	DB	DOC- SHELL	EDITOR	LAYOUT	MEMORY	MODU- LES	SECU- RITY	STO- RAGE	TOOLKIT	URL- LOADER	WIDGET	GFX	SPELL- CHECKER	NSPR	XP- CONNECT	OTHERS
NETWORK	0.00	162.41	0.06	331.03	4068.96	4.43	0.00	400.66	0.00	2.77	4156.75	618.17	13.56	69.78	12635.35	91.58	139.85	0.01	0.00	103946.84	0.00	60.07
JS	14494.90	0.00	0.00	92299.21	5.06	0.13	0.00	288.98	108.43	3784.87	54286.31	69.04	21.01	35.69	360.53	3.41	2.06	0.00	0.02	535709.15	2148.68	36311.77
PARSER	19.06	0.00	0.00	2279.28	211.94	0.00	0.00	0.00	0.00	0.09	562.52	0.62	0.00	0.00	1295.93	0.00	46.22	0.00	0.00	10575.30	0.00	894.18
DOM	5562.20	41075.31	4847.44	0.00	8496.10	14.64	0.00	5061.64	46928.79	117802.40	8917.99	3429.16	4.63	8.39	6851.23	1.48	1175.09	3987.79	73.13	1463654.67	0.00	68354.94
BROWSER	11.72	12.69	0.00	20.57	0.00	0.00	0.00	19.35	0.00	0.00	126.95	0.01	0.00	0.00	1160.68	1.43	0.00	0.00	0.00	1933.34	0.00	4.47
CHROME	30.52	0.00	0.00	0.00	8.21	0.00	0.00	0.00	0.00	0.00	0.00	1.79	0.00	0.81	0.00	0.00	0.00	0.00	0.00	36.81	0.00	0.37
DB	0.00	0.00	0.00	0.00	68.84	0.00	0.00	0.00	0.00	0.00	2141.20	0.00	0.00	110.16	4.90	0.00	0.00	0.00	0.00	222.41	0.00	0.00
DOCSHELL	1156.03	0.00	0.00	2190.07	281.78	0.00	0.00	0.00	0.04	1109.05	27.66	791.10	2.81	0.00	84.84	11682.67	50.60	0.00	0.00	20715.47	0.00	320.51
EDITOR	356.37	0.00	0.00	6333.11	1390.62	0.00	0.00	1.17	0.00	402.55	1544.83	6.72	0.00	0.00	8371.63	0.00	3.75	0.00	12.27	15000.11	0.00	2483.37
LAYOUT	3286.57	265.57	0.47	4152796.98	1391.53	0.00	0.00	354.23	144553.64	0.00	12700.40	3401.06	0.00	0.00	5772.31	0.00	8790.76	366906.38	0.00	458985.23	0.00	69545.22
MEMORY	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	38.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	99.62
MODULES	3041.96	0.02	0.00	4330.58	1132.13	0.00	0.00	88.86	0.00	4947.62	1144.55	0.00	0.59	0.00	1297.21	0.28	20.94	9620.44	0.14	36766.37	0.00	87789.49
SECURITY	426.83	1.43	0.00	369.78	2.20	0.00	6.94	6.76	0.00	0.00	570.16	337.14	0.00	0.00	316.87	47.55	8.80	0.00	0.00	6413.26	0.00	2.77
STORAGE	0.00	19.30	0.00	0.02	6.20	0.00	2010.70	0.00	0.00	0.00	62.22	0.25	0.00	0.00	106.48	0.00	1.72	0.00	0.00	1124.51	0.00	66.53
TOOLKIT	151.74	4.07	0.00	30.19	2590.97	0.00	994.70	19.28	0.00	7.64	2154.27	2.63	6.36	1335.67	0.00	0.09	6.50	0.09	0.02	3102.25	0.00	11507.93
URLLOADER	190.50	11.46	5.31	39.05	386.66	0.00	0.00	1862.87	0.00	0.00	97.89	172.94	239.04	0.00	62.70	0.00	0.00	0.00	0.02	18778.90	0.00	660.79
WIDGET	62.09	850.71	19.50	4064.55	345.50	0.00	0.00	1.45	0.00	4981.68	42962.61	681.74	0.30	1.34	824.29	0.00	0.00	6349.75	0.62	29010.15	0.00	281.28
GFX	0.70	0.00	0.00	464.46	1248.44	0.00	0.00	0.00	0.00	2399.08	39676.85	55.88	0.00	0.00	2592.21	0.00	268192.63	0.00	171532.58	0.00	473803.75	
SPELL- CHECKER	0.26	0.00	0.00	397.69	0.36	0.00	0.00	0.00	27.80	18.93	1413.69	0.33	0.00	0.00	716.08	0.00	1.47	0.00	0.00	721.78	0.00	959.87
NSPR	12.85	62.16	1616.44	218.76	0.00	0.00	0.00	0.00	0.00	64.57	6643.21	0.00	366.46	0.88	2.16	0.00	1241.48	0.00	0.00	0.00	0.00	0.48
XPCON- NECT	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
OTHER	6047.34	262329.05	0.12	77071.65	951.96	0.00	0.00	2322.61	440.51	45662.77	1058.81	1434.29	0.90	1.16	1709.56	0.30	4708.42	23138.19	0.00	45496.57	13358.78	0.00

Table 4.4: Number of Calls (in 1000s) across Code Units over 100 Websites This table lists the number of calls from the row component to the column component.

	NET- WORK	JS	PARSER	DOM	BROW- SER	CH- ROME	DB	DOC- SHELL	EDITOR	LAYOUT	MEMORY	MODU- LES	SECU- RITY	SIT- RAGE	TOOLKIT	URI- LOADER	WIDGET	GFX	SPELL- CHECKER	NSPR	XPCON- NECT	OTHERS
NETWORK	0.00	904.18	134.77	4191.41	33.75	18.29	154.20	477.31	737.36	641.86	30.45	2363.08	146.31	132.92	2837.50	108.34	200.17	418.25	2.55	4055.53	0.00	1262.92
JS	2649.79	0.00	5276.54	19093.42	43.30	0.66	611.05	132.55	2265.96	2877.13	462.99	14398.82	559.72	11.75	1269.94	16.58	89.47	4661.86	592.32	587.86	19.63	27847.73
PARSER	172.66	724.94	0.00	1349.52	22.99	0.33	45.81	4.03	820.26	364.78	7.66	901.46	105.02	0.07	11.52	2.14	29.38	281.73	19.86	73.11	0.00	826.42
DOM	842.87	38170.49	688.51	0.00	345.24	0.78	104.14	270.64	4737.52	96021.06	143.93	1593.42	200.15	0.74	756.84	14.06	33830.68	98291.55	262.40	922.63	0.00	9815.02
BROWSER	6414.98	15.98	190.06	1783.02	0.00	4.38	2.42	15.25	1199.77	1697.25	5.54	1635.10	10.07	2.16	2310.87	1.41	23.99	463.48	0.28	10.53	0.00	1530.87
CHROME	2.91	0.00	0.05	0.36	0.05	0.00	0.00	0.00	0.04	0.35	0.06	0.34	0.00	0.00	0.45	0.00	0.00	0.01	0.01	0.70	0.00	0.35
DB	127.60	234.91	18.10	115.17	1.41	0.03	0.00	5.72	41.57	56.74	23.26	111.20	22.33	14.35	19.03	0.85	5.34	136.23	2.66	26.71	0.00	82.27
DOCSHELL	113.84	14.41	0.19	610.73	11.72	0.01	9.82	0.00	2.05	109.70	4.35	28.01	3.16	0.00	11.54	69.92	1.77	8.41	0.06	77.99	0.00	189.45
EDITOR	8271.92	2598.52	86.75	6711.58	438.47	0.18	3.74	7.17	0.00	7663.72	13.65	103.08	1.70	1.19	209.38	1.46	0.90	254.67	22.90	391.10	0.00	1175.65
LAYOUT	688.80	3268.52	356.68	76185.30	378.59	0.12	113.30	97.48	5822.37	0.00	124.80	3932.96	125.70	0.65	1114.43	3.67	4861.88	36232.96	80.71	1604.38	0.00	4019.22
MEMORY	36.03	236.65	12.19	174.43	0.06	0.00	372.71	3.64	4.59	382.43	0.00	73.69	144.63	0.83	597.35	1.26	1053.96	8511.56	1.30	147.29	0.00	484.44
MODULES	1320.91	14253.65	2867.16	3551.74	24.48	2.17	166.48	46.83	652.08	1928.00	80.43	0.00	284.90	1.40	611.35	28.67	1811.32	39609.94	494.46	413.53	0.00	45803.64
SECURITY	68.91	118.93	17.15	50.88	0.76	0.01	148.79	6.16	19.29	33.30	7.09	147.51	0.00	0.17	66.40	52.42	22.52	65.96	0.52	136.22	0.00	135.87
STORAGE	118.04	5.33	0.11	2.15	0.05	0.00	10.75	0.59	0.93	0.33	0.03	0.25	0.12	0.00	66.86	0.00	1.34	0.58	0.00	26.98	0.00	1.09
TOOLKIT	276.24	135.39	1747.75	2876.74	287.31	0.68	7.11	52.27	4688.79	1879.02	17.51	72.52	15.62	8.02	0.00	2.84	43.60	351.50	16.13	47.12	0.00	834.30
URILOADER	10.91	13.46	0.44	7.27	0.32	0.08	3.42	248.60	1.99	2.63	0.37	12.83	0.81	0.02	2.47	0.00	0.23	4.58	0.02	4.31	0.00	40.83
WIDGET	179.26	45.21	6.53	145.77	1.07	0.00	3.50	10.58	17.61	195.54	4.67	42.88	9.58	1.23	11.37	1.61	0.00	9458.30	0.42	237.34	0.00	196.98
GFX	594.09	2036.95	401.69	18668.68	246.74	0.16	105.41	81.82	571.79	32639.04	138.49	37246.92	94.05	0.31	296.00	7.56	18176.48	0.00	39.39	80.00	0.00	5035.49
SPELL- CHECKER	1.93	5.54	1.10	6.62	0.14	0.01	1.20	0.06	22.41	12.51	0.16	6.63	0.10	0.00	0.59	0.01	0.84	4.56	0.00	0.84	0.00	27.65
NSPR	771.43	359.36	127.36	695.08	45.96	0.06	60.98	13.19	94.10	4256.19	199.74	257.23	286.44	3.95	17.57	2.37	485.43	1135.26	1.79	0.00	0.00	192.67
XPCON- NECT	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
OTHER	631.68	47050.07	16718.05	9582.00	330.33	0.59	99.58	81.93	2009.31	2152.58	64.61	170510.13	95.89	1.16	328.14	113.36	164.82	63652.66	176.85	178.86	0.00	0.00

Table 4.5: Exchanged Data Size (in kilobytes) This table lists the bytes of data exchanged from the row component to the column component.

Cross-Origin Access		Number of Calls	Data Size of Calls (KB)
Browser Side	postMessage	4,031	587
	location	9	-
	window.parent	24	-
	window.frames	3,330	-
Network sub-resource	Images, CSS, etc.	10,745	131,920

Table 4.6: Cross-Origin Calls & Sub-Resource Loading

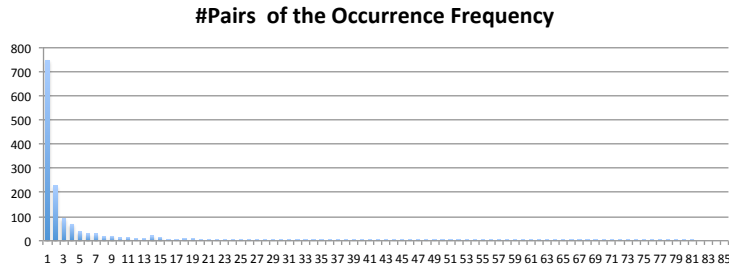


Figure 4.3: Occurrence Frequencies of Unique Pairs of Different Requestor-Destination Origins 746 unique pairs only occur once, while only 164 unique pairs occur more than 15 times.

To better illustrate these results, Figure 4.2 depicts different volumes of inter-component functions calls with gray scales. Darker colors indicate heavier communications between the corresponding components.

Results of M4. Table 4.5 lists the measured data exchange sizes between components. For example, the DOM and the Layout engine have larger data exchange than other components: 172,206.36 Kilobytes over the 100 websites.

Measuring M5 & M6. We intercept the calls to client-side communication channels in Firefox, retrieve the caller and callee origins, and record the size of data passed in `postMessage` calls.

Results of M5. Table 4.6 summarizes the number of client-side calls to access other origins.

Results of M6. *Size of data exchange in client-side cross-origin calls.* Table 4.6 also summarizes the size of data exchanged in client-side calls (passed in `postMessage` calls).

Measuring M7. We intercept all network responses to Firefox, and identify whose requester and destination origins are different. We record such cases with the size of data passed in the HTTP response body.

Results of M7. To evaluate in more detail the performance implications in using separate partitions for sub-resource loading, we measure the number of cross-origin sub-resources for each of the Alexa Top 100 websites. Table 4.6 summarizes the overall

Size of Message (in bytes)	Average RTT for Unix Domain Socket	Average RTT for Network Comm	Average RTT for Cross-VM Communication
50	4673	87642	252008
500	5045	176160	288276
1000	5145	276841	252107
2K	5821	367356	251605
4K	6838	449262	269845
8K	9986	638598	336999

Table 4.7: Round-Trip Time (RTT) of Unix Domain Socket, Network and Cross-VM Communications, *in nanoseconds*, Averaged over 10,000 Runs Each

call numbers of size of data exchanged during cross-origin sub-resource loading. In our measurement, the largest number is 51, with `www.sina.com.cn`. Figure 4.3 shows that the re-occurrence rate of unique pairs of different requester and destination origins is very small. More than 746 pairs occur only once. In fact, there are in total 1,515 such unique pairs, averaged to $1,515 / 100 = 15$ pairs for each page.

For Goal 3: isolation primitive overhead. We also measure performance overhead with different isolation primitives, in communication cost **M8** for Performance Parameter **C6**, and in memory consumption **M8** for Performance Parameter **C7**.

M8: *Communication delay with sockets, network and VMs.*

M9: *Size of memory consumption of a partition of a process, or an VM.*

Measurement & Results of M8. We use a simple client-server communication program to measure the inter-partition call costs between Unix domain sockets, between hosts connected via LAN, and between virtual machines on the same VM host. We average over 10,000 rounds of each primitive with varying message lengths. Table 4.7 summarizes our measurements on round trip times for inter-partition communications with the three isolation primitives. Unix domain sockets is 6-10 times more efficient than cross-VM communications.

Measurement & Results of M9. By checking the size of empty process on different hosts, we conclude that the memory used by an almost-empty process is about 120k-140K on Ubuntu with the `pmap` utility. As this number was very stable across different runs, we take this as the memory consumption of creating processes. For the Ubuntu guest OS we created, the write-able/private memory used by VirtualBox was about 25M bytes and the memory used by guest OS running in VirtualBox was about 90M bytes. We take 90M as the size of VM partition memory cost, and 25M as the memory overhead from a VM daemon in our quantification. Therefore, a Linux process incurs $90M / 130K = 709$ times lower memory overhead than a VM.

4.4.3 Summary of Findings

In this section, we summarize the high-level findings from our detailed measurements. Specifically, we revisit the partitioning dimensions outlined earlier and evaluate their

Partitioning Dimension	#Vulnerabilities Mitigated	#Lines of Code Partitioned	Comm Cost	Data Exchanges Cost	Memory Cost
Single Process	0	N.A.	0	0	0.13K
One Process per Origin (w/o Cross-Origin Sub-Resource Isolation)	0	N.A.	0	0	130K
One Process per Origin (with Cross-Origin Sub-Resource Isolation)	38	N.A.	0.37ms	5.87MB	1.4MB
One Process per Pair of Requester-Destination of Sub-Resource	38	N.A.	0.91ms	7.19MB	2.1MB
Renderer/Browser Division	81	1,863K	2.59min	3.54MB	130KB
JS/DOM Separation (Process)	147	JS:367K DOM:155K	6.67s	572.6KB	130KB
JS/DOM Separation (Network)	147	JS:367K DOM:155K	3.78min	572.6KB	125MB
Layout/Window Manager (GFX+Widget) Separation	69	Layout:367K GFX+Widget:615K	19.15s	739.3KB	130KB
DOM/Layout Separation	102	DOM:155K Layout:367K	3.56min	1.68MB	130KB

Table 4.8: Security Benefits and Performance Costs of Partitioning Dimensions *Performance costs are per page, averaged over Alexa Top 100 websites.*

security-performance trade-offs. We also summarize the performance bottlenecks that our measurements highlight.

Table 4.8 summarizes the security and parameters for each design point along the dimensions being debated in present designs. The values in the table for performance costs are per web page, averaged over the Top 100 Alexa pages.

Origin-based isolation. One process per origin without separating cross-origin sub-resources have no security benefits. If contents from another origin hosted as sub-resources (such as PDF) can still be processed in the same partition, security vulnerabilities can still permit unintended escalation of privileges. This is consistent with the observations made by several browser designs which propose hosting sub-resources in separate containers, as we explain below.

Sub-resource Isolation. Several browsers propose isolating each pair of requester-destination of sub-resources to be further isolated in separate partitions, such as Gazelle [169] and IBOS [154]. Our data suggests that such a design mitigates CROSS-ORIGIN vulnerabilities, but has a large performance cost. For instance, the memory cost of creating several partitions (using processes) is large and will be a performance bottleneck. In our measurement, one web page can include up to 51 third-party sub-resources. If all these cross-origin sub-resources are to be isolated by different processes, and consider that a typical browser process would need 20 Megabytes [5], then around 1 Gigabyte memory overhead will be incurred just for loading third-party resources for this single web page. Therefore, although sub-resource isolation can mitigate 38 CROSS-ORIGIN vulnerabilities, browsers may need to optimize memory usage for processes that load sub-resources before they can practically adopt this proposal.

It is interesting to compare our identified bottlenecks to choices made by today's

web browsers. For instance, Google Chrome does not suffer from this performance bottleneck by making a security-performance trade-off. It adopts a different strategy by grouping resources according to a site-instance [35] of the hosting page, which significantly reduces the number of processes created [35]. We leave the detailed definition and discussion of this strategy out of scope; however, we believe that our methodology does identify realistic practical constraints.

Component-based isolation. Isolation by components mitigates WEB-TO-COMP vulnerabilities. For example, the JavaScript engine and the DOM have 147 such vulnerabilities. At the same time, the 367K of source code (TCB) in the JavaScript engine can be isolated, which is 10% of the entire browser. Nevertheless, since they have frequent interactions, such isolation costs prohibitively high communication and memory overhead. Hence, although beneficial for security, such a partitioning dimension is less practical for adoption. For instance, designers of OP redacted the decision to isolate JavaScript engine and the HTML parser within one web page instance in OP2; our measurement identifies this high overhead as a bottleneck in Table 4.4.

Renderer/Browser kernel isolation. We also take a popular architecture of renderer/browser kernel division for evaluation. We evaluate our methodology on the Google Chrome design model to measure the security benefits and performance costs. Such a partitioning dimension would prevent WEB-TO-COMP vulnerabilities in the renderer process, and WEB-TO-SYS vulnerabilities. If we apply Firefox code size to this design, the size of TCB in the kernel process would be around 1,863K, i.e., 53.5% of the browser codebase. Note that this is just a rough estimation based on our blueprint of coarse-grained components. Further dividing components can reduce the necessary code size that needs to be put into the browser kernel process.

Our measurements identify potential performance bottlenecks that correlate with actual browser implementations. Specifically, we find that isolation between components in the renderer processes and the browser kernel process, as in Chrome, would incur very high performance overhead, such as between the GFX and the Layout engine. However, such performance bottlenecks do not appear in Chrome. We observe substantial effort [146] with Chrome that is relevant in optimizing render-kernel communication performance. Besides, Chrome also uses GPU command buffers and other optimization techniques to improve performance of rendering and communication [128].

Component partitioning with high security benefits. We identify a few browser components that have high security benefits to be isolated from other components. For example, the JavaScript engine is a fairly complex component with 367K lines of source code, and 88, i.e., 31.8% of, WEB-TO-COMP vulnerabilities. Isolating it from other browser components will mitigate a large fraction of vulnerabilities. Other typical example components include the Layout engine with 367K lines of source code and 43 (15.5%) WEB-TO-COMP vulnerabilities, as well as GFX, the rendering component for

Firefox, with 478K lines of source code and 22 (7.9%) WEB-TO-COMP vulnerabilities.

Component partitioning with high performance costs. We identify the main browser components that have tight interactions with other browser components. Thus, isolating them from others would incur high performance costs. For example, as shown in Table 4.4, our measurements find 133,374,520 function calls between the JavaScript engine and the DOM, and 369,305,460 calls between the GFX rendering engine and the Layout engine. To show why they can become performance bottlenecks, here is a simple calculation. Suppose they are separated by processes, a single RTT with Unix domain sockets costs around 5000 nanoseconds delay. If there is no additional optimization in place, these numbers correspond to $133,374,520 * 5000 \text{ nanoseconds} / 100 \text{ pages} = 6.67 \text{ seconds/page}$ and $18.47 \text{ seconds/page}$, respectively. Such performance overhead is prohibitively high. Security architects should either avoid such partitioning, or take further measures to optimize these performance bottlenecks.

4.4.4 Discussion

Our identified bottlenecks provide data-driven insights into which parts of browser design need careful engineering. As we have discussed earlier, careful engineering and implementation-level design can mitigate these bottlenecks. Below we discuss the scope where our results are applicable and point out some alternative choices a security analyst can make while re-applying our methodology to obtain potentially different results.

Finer-grained blueprint. In our experiments, we use the blueprint based on major browser components. A security architect can easily re-configure the boundaries or granularity of the definitions of code unit. We foresee that applying our methodology to finer-grained blueprints may yield other interesting results in guiding browser designs. For example, blueprints on source code classes or functions may help developers redraw the boundaries of browser components.

Applying other isolation mechanisms to browsers. Our methodology is focused on isolation based on OS primitives, host isolation and VMs. This is because most prior works on applying privilege separation to browser redesigning have focused on isolation primitives based on processes, VMs, and hosts, etc. There is another line of isolation mechanisms based on dynamic taint analysis [116], software-based isolation [167] and so on. We expect that modifications to our methodology will be necessary to accommodate these in the future, though several of our insights will continue to hold. We hope further researches will explore these possibilities to quantify privilege-separated designs in web browsers.

4.5 Related Work

Privilege separation. The concept of privilege separation in computer systems was proposed by Saltzer et al. [142]. Since then it has been used in the redesign of several legacy OS applications [131, 18] (including web browsers) and even web applications [51, 5, 13]. Similar to PRIVGAUGE’s goals, several automated techniques have been developed to aid analysts to partition an existing application, such as PrivTrans [25], Jif/Split [184], and Wedge [22]. Most of these works have focused on the problem of privilege minimization, i.e., inferring partitions where maximum code executes in partitions with minimum or no privileges, while performance is measured “after-the-fact”. Our work, in contrast, aims to quantify performance overhead with privilege-separated designs with only a blueprint without the actual implementations. Our work also differs with them by performing measurements on binary code, rather than source code.

Privilege separation in browsers. Our work is closely related to redesign of web browsers, which has been an active area of research [111, 15, 36, 169, 62, 63, 75, 154, 92]. Our work is motivated by the design decisions that arise in partitioning web browsers, which performs a complex task of isolating users, origins and the system. Among them, IE uses tab-based isolation, Google Chrome [15] isolates web origins into different renderer processes, while Gazelle [169] further isolates sub-resources and plugins. Our measurements have shown that some web pages may include 51 sub-resources of different destination origins. Our data quantifies the number of partitions that may be created in such designs as well as in further partitioned browsers, such as OP [62] and OP2 [63]. In addition, our measurements also evaluate the performance costs in VM-based isolation, such as Tahoma [36], and memory consumption from separate network processes for sub-resources in IBOS [154] design. Our work advocates privilege-separated browsers for better security, and identifies potential performance bottlenecks that need to be optimized to trim their performance costs.

Evaluation metrics. Estimation of security benefits using bug counts is one way of quantifying security. Riscorla et. al. discuss potential drawback of such reactive measurement [137]. Other methods have been proposed, but are more heavy-weight and require detailed analysis of source code [94, 71, 138]. Measurement of performance metrics such as inter-partition calls and data exchange has been identified in the design of isolation primitives such as SFI [167]. We provide a comprehensive in-depth empirical analysis of these metrics in a widely used web browser (Mozilla Firefox).

4.6 Summary

In this work, we propose a measurement-based methodology to quantify security benefits and performance costs of privilege-partitioned browser designs. With an assistance tool PRIVGAUGE, we perform a large-scale study of 9 browser designs over Alexa

Top 100 websites. Our results provide empirical data on security and performance implications of various partitioning dimensions adopted by recent browser designs. Our methodology will help evaluate performance overhead in designing future security mechanisms in browsers, and we hope this will enable more privilege-separated browser designs to be adopted in practice.

Chapter 5

BSM: Towards General Security Support in Browsers

In the previous two chapters, we propose a new primitive for transparent isolation of untrusted JavaScript, and a methodology for evaluating security and performance implications for different program partitioning schemes in browser design. They enable more effective and efficient isolation solutions in the browser environment. However, there are additional requirements for principled security in browsers. After a carefully chosen program partitioning, within the same partition, we need additional mechanisms to monitor and control web application behaviors. Moreover, such a behavior control framework may also allow browser designers to quickly test with different partitioning schemes by applying access control on scripts and resources with different identities. We thus propose a general security framework to achieve intra-partition behavior monitoring and control.

Traditional web security mechanisms, such as the same-origin policy (SOP) [113] and cookie-based session authentication, are no longer sufficient to secure web applications in the complex browser environment, which often involves third-party JavaScript libraries, web mashups, and Internet advertisements inside the same origin. As a result, many web-based attacks have emerged, leveraging the complex execution environment and layout of web applications. Examples include cross-site scripting (XSS) [120], cross-site request forgery (CSRF) [14], and privacy violation [81].

To mitigate these threats to web applications, a wide range of solutions have been developed [158, 37, 105, 20, 67, 88, 68, 83, 162]. The focuses in these solutions are specific security enhancement or mitigation. However, the fundamental reason for the ever-emerging web application attacks is that the existing security mechanisms in browsers cannot cope with the browsers' new role as a complex operating environment for rich Internet applications.

There are also proposals for enhancing browsers' security mechanisms. For example, the recent trend of including executable contents from external parties demands additional isolation and access control within a hosting page. MashupOS [168] addresses this problem by providing new abstractions designed for different trust scenar-

ios in mashup applications. As another example, ESCUDO [82] adopts a ring-based security model to regulate JavaScript's accesses to different regions of the page DOM. Both solutions are achieved through browser modification.

There has been no systematic analysis for general security support that is required to enforce flexible security solutions in browsers. Such an analysis will give us a clear picture on how to systematically enhance web browsers, so that instead of changing the web browser for a specific type of attacks, we can build general security support into the web browser. With such support, new security proposals can be readily developed and adopted without directly modifying the browser itself.

In this work, we systematically analyze the requirement for a flexible security framework in web browsers, aided by the reference model of the browser presented in Chapter 2 (Figure 2.1). Following the traditional security principles in operating systems [142], we identify several critical security requirements needed in web browsers. We then propose a general security framework, *Browser Security Modules (BSM)*¹, to incorporate these requirements into web browsers.

BSM is designed to be *general*, *flexible* and *extensible*. A *general* design provides interfaces to implement security solutions that are not specific to any particular web browser or any particular attack. Once a solution is developed, it can be applied to many web browsers. It is *flexible* as it provides the necessary interception points with essential security information, and allows security solutions to use their own logic. Moreover, we understand that in certain special cases, a security solution may require uncommon functionality that is not generally needed by other security solutions. Therefore, our security framework also provides interfaces to *extend* its functionality according to security solutions' needs.

To demonstrate its feasibility, we have prototyped BSM into Firefox 3.5 and 8.0. We show how BSM supports the functionality required by existing browser-based solutions for web application security. Our preliminary evaluation on performance show that BSM itself incurs negligible performance overhead, and a security module with a reference implementation of ESCUDO [82] has reasonable performance overhead.

In addition, we also show that BSM can be used to monitor HTTP request dependency and the evolving states of web applications to detect sophisticated attacks.

5.1 Requirements for Flexible Security in Web Browsers

In this section, we identify the general security requirements needed by browsers to regulate run-time behaviors of web applications. Note that our analysis is not specific to any attack. Instead, we analyze the underlying mechanisms of existing attacks to discover the weaknesses of the existing protection mechanism. We then follow traditional OS security principles [142] to identify the security requirements in supporting enhanced web application security.

We focus on threats to the web platform that can be mitigated by security mecha-

¹The name BSM is inspired by Linux Security Modules (LSM) [176].

nisms in the browsers, as discussed in Section 2.2.1. Therefore, browser-based attacks to the operating system, such as drive-by download attacks, are not in the scope of our work. The solutions to such problems are orthogonal to our approach [36, 62, 15, 169, 154].

First of all, general and flexible security support would require the web browser to mediate sensitive operations of web applications running in it. Specifically in an event-driven model of modern web applications, client-side events occurring in the browser environment are essential triggers to sensitive actions, thus also need to be monitored by the browser. For example, the browser needs to intercept the click event that would subsequently trigger the deletion of an email in a webmail system. With such interception, the browser also needs to have finer-grained principals than origins to mount any flexible control. Otherwise, behaviors of web mashups that are included into the same origin cannot be regulated distinctly. Finally, the browser needs to provide sufficient information for any security solution to discern the legitimacy of requested accesses from web applications.

We thus summarize the requirements as follows.

- *Requirement 1: Intercepting sensitive operations and events in the browser.* We need to intercept security-sensitive operations and events in browsers, so that web applications can enforce additional checks and regulations on cross-principal access. For example, this can be used to isolate the execution of untrusted scripts and strictly monitor and control their accesses to other resources in the web application. Moreover, we also need to provide the capability to abort or alter the execution of operations intercepted.

To identify the set of security-sensitive operations, we must examine the browser internals and their components carefully. The interception should cover the behaviors of interactions among major browser components, including the network module, the JavaScript engine, the parsers for HTML, CSS, URIs, etc., as well as the Document Object Model (DOM). There is also another level of operation interception inside the JavaScript engine [124], while in this work we focus on the operations between major browser components.

- *Requirement 2: Flexible and finer-grained principals in the browser.* SOP uses origin, defined as <protocol, host, port>, to separate resources in the browser. Recent threats from XSS and malicious mashups have already called for further division of resources included in one origin. We need to provide web applications the ability to assign fine-grained principals to components, so that they can accurately control the interactions among different components, with the flexibility to update the principals at runtime. This is also the basis to achieve the *least privilege principle* for scripts.

Scripts can insert or modify components in web applications, such as inserting a new piece of script in DOM element attributes. Such dynamic features are

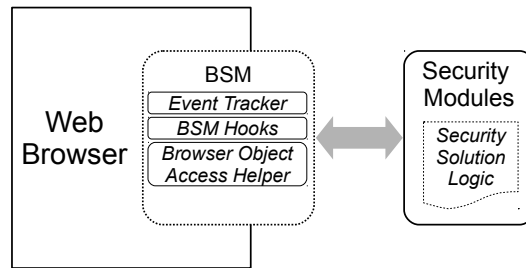


Figure 5.1: The Architecture of BSM

required by modern web applications. We need to track the effective principals of components to accurately decide the access.

- *Requirement 3: Extracting context information and browser state.* Modern web applications are event-driven. To accurately understand their behaviors, web browsers need to maintain the causal relationships between triggering events and actions, such as request initiation, DOM mutations, etc. Such information is also required to discern sophisticated attacks that attempt to mimic normal behaviors. Security mechanisms should be able to examine necessary browser states to determine the nature of behaviors intercepted. This includes the details of the events and other browser objects relevant to HTTP requests.

Many existing security solutions can be built upon the security support we discuss here, as we will analyze in Section 5.3.3. In addition, with such security support, we are also able to develop other types of security solutions. For example, with the interception over sensitive browser operations and events, we can monitor the execution of web applications to detect malicious behaviors according to the runtime dependency between events.

The requirements discussed above outline the crucial aspects in supporting flexible security solutions in web browsers. However, admittedly, they do not constitute a complete list of all requirements for web browser and application security in general. We demonstrate that by supporting such requirements in the browser environment, we enable effective security solutions. On the other hand, we advocate principled analysis of other security requirements, such as trust management, verified execution, etc., for the browser platform.

5.2 Design of Browser Security Modules

To support the security requirements we have identified, we present the design of a general security framework for web browsers: Browser Security Modules (BSM), with design goals of *general security support*, *flexibility*, and *extensibility*.

Category	Example Interfaces
Network Communications	httpchannel_init, process_response, process_redirection
Parse Content	parser_read_buffer, parse_web_content, handle_token
Construct DOM	create_html_element, set_principal, get_principal
Load Resources	scriptloader_load, frameloader_load, imageloader_load
Process Scripts	scriptloader_process_script
JavaScript's Access to DOM	js_dom_call, document_getelementbyid, document_cookie, window_open, cookie_obj_access
Register to Events	dom_addjseventlistener, js_add_event_listener
Invoke Event Listeners	js_call_event_handler
XMLHttpRequest	xmlhttprequest_open, xmlhttprequest_send
Capture & Process Events	event_manager_handle_event, checkbox_selected, input_entered, button_clicked, etc.
Plugins	plugin_instantiate

Table 5.1: Example Interfaces Defined in BSM Hooks

5.2.1 BSM Architecture

Figure 5.1 illustrates the architecture of BSM. It contains four major components: BSM Hooks, the Event Tracker, the Browser Object Access Helper, and browser-independent Security Modules that implement security solutions.

BSM Hooks intercept security-sensitive operations and events in the browser. After the sensitive operations or event handlers in the browser are intercepted, the functions registered on the hooks, which is in Security Modules, are invoked to implement additional security mechanisms. When a function in a security module returns to the BSM Hook in the browser, it can optionally abort or alter the current operation according to the return value from the security module.

The Event Tracker is a component that provides additional dependency information among events for security applications. The Browser Object Access Helper is a browser-specific module that understands the internal data structures of specific browsers, and provides helper functions for accessing browsers' custom objects to security modules.

We explain how the security requirements we identified are supported in this design. Requirement 1 is met by BSM Hooks, and Table 5.1 lists some examples of the interfaces defined by BSM. Next, we describe how other BSM components supports the rest two requirements identified in Section 5.1.

5.2.2 Fine-grained Principal Support and Tracking

To support Requirement 2, BSM keeps track of fine-grained principals, such as intra-origin principals, and provides APIs to access them, i.e., `set_principal` and `get_principal`. `get_principal` allows security modules to obtain principal information from the DOM elements, and store it internally, while `set_principal` allows them to update the principals of DOM elements. These interfaces allow security modules to track the identity of DOM elements, which may be updated during runtime.

In BSM Hooks where a new DOM element could be created, they receive an additional argument `propagate_principal`. When this argument is set to `true` by security modules registered to the hook, BSM propagates the current principal of the subject element to the object element created.

5.2.3 Accurate Browser States and Context Information

The interception of an operation and identification of its principal provide the basis of implementing new security mechanisms in security modules. However, the power of such mechanisms is largely dependent on how Requirement 3 is met. For example, given an operation, security modules need the access to details of the operation, such as the arguments of the current function call and the events triggering this operation. Based on such browser environment information, a security module knows whether a request is triggered by a user action or by JavaScript, which provides the key insight in distinguishing legitimate and malicious behaviors of injected scripts.

Providing accurate browser states and context information in the dynamic browser environment is challenging. Modern web browsers are event-driven and multi-threaded. Therefore, events may occur asynchronously in web browsers. The chronological order of event occurrences is not suitable for security checking and regulation, as the order may vary in different runs. Rather, BSM needs to provide the exact dependency relationships among browser internal events to build reliable detection models. For example, when an HTTP request is generated, instead of telling that it is preceded by a user click, we need information such as it is triggered by a user click, or by a JavaScript timer, etc.

Besides, function arguments or local variables in the original browser function calls are typically of custom types defined by the browser, so another challenge is to hide specific browser implementation details in BSM Hooks to make security modules generally applicable, but still make necessary data accessible to them for effective examination on the ongoing browser operations.

BSM solves these two challenges with the Event Tracker and the Browser Object Access Helper, respectively, as detailed below.

Tracking context of browser operations The Event Tracker is used in conjunction with BSM Hooks to track the events currently being processed. When an event occurs and is processed by the web browser, we register the event with the Event Tracker. When the browser finishes processing the event, we de-register the event with the Event Tracker. When registering events with the Event Tracker, we also pass important parameters with it, such as the subject and object of the event. The security module will extract information from the parameters using the Browser Object Access Helper, convert them into standard data structures, and construct a new Event Tracker entry. Each entry is composed in the form of

<Event Type, Subject Reference, Object Reference, Attribute References>,

where the first element is compulsory and the latter three are optional. This way, the Event Tracker keeps track of the dependency relations between currently occurring events.

Nevertheless, such instantaneous dependency relations are not sufficient, as it does not provide any information regarding events that occurred previously and have been finished processing but still affect the current events. Sometimes security applications

also need the dependency relations at different points of time. For example, after an HTTP channel is initialized, the function will return, and the browser will send an HTTP request at a later time using this channel. However, when it is actually about to send the HTTP request, the event of the channel initialization has been de-registered with the Event Tracker. As a result, at that time, we would have no idea how the HTTP request was generated. Another example is that JavaScript functions on web pages are always evaluated whenever the parser finds them, but JavaScript functions may be invoked at a later time scheduled by timers, or triggered by asynchronous events such as HTTP response arrivals or mouse clicks. Hence, when they are actually executed, the Event Tracker would have no context information when they were first inserted into web pages.

To overcome such limitations, the Event Tracker correlates additional dependency relationships at different points of time, according to the targets of the events. For the HTTP channel example, although the initialization and request sending are two asynchronous events that typically occur at different points of time, they operate on the same HTTP channel object, and thus can be correlated with it. Similarly, the evaluation and invocation of a JavaScript function can also be correlated with the same JavaScript function object. By such additional correlation, the Event Tracker extends the dependency relations of one event to its correlated events.

Security modules can select the events they want to track with the Event Tracker, while in our current prototype, we focus on user interactions, timers, and important actions such as HTTP requests, DOM mutations, etc.

Accessing browser objects When invoking security modules, each BSM Hook will pass a list of parameters to them. It includes important parameters in the original browser function call as well as local variables that are commonly required by security applications. However, such parameters are of the custom types defined by the browser. By design, security modules are *general* to all browsers, so they must be independent of a specific browser implementation, thus independent of browser data type definitions. To facilitate security modules to access the parameter objects, we introduce the Browser Object Access Helper that provides access to browser object types. The Browser Object Access Helper is a browser-specific module that understands the internal data structures of specific browsers, and provides helper functions for accessing browsers' custom objects to security modules.

For the list of parameters passed by BSM Hooks to security modules, we select function arguments and local variables that are useful in security applications we need to support in our case studies. They are expected to cover the needs by most security solutions. However, to provide further extensibility, we pass an additional parameter of the `this` pointer from the interception point to security modules.

The Browser Object Access Helper also allows security modules to overwrite the values of browser objects through parameters or the `this` pointer, although such modification needs to be done with special care as it may cause unpredicted browser errors.

5.3 Experiments and Evaluation

To evaluate our BSM design and its compatibility with existing web browsers, we implemented prototypes of the security framework in Firefox 3.5 and Firefox 8.0 on Linux. Our code is mainly written in C++, together with some utilities in python. The goal of our evaluation is to demonstrate the feasibility and preliminary performance results. We will discuss our plan on guaranteeing the completeness of the BSM implementation in Section 5.4.

5.3.1 Implementation

Table 5.2 summarizes the functions we have intercepted and SLOC of the hooks added into the browser source code and sample security modules.

Version	#Hooks	SLOC of Hooks	SLOC of Security Module
Firefox 3.5	476	11,639	15,013
Firefox 8.0	39	1,065	1,506

Table 5.2: Summary of Hooks and SLOC

Firefox 3.5 was our first experimental platform, and we worked in a conservative way where we tried to hook any function that is related to security. The security module in Firefox 3.5 prototype basically serves as a daily experimental platform to test new web security research ideas. Thus, it has more hooks and security module logic than the prototype on Firefox 8.0. However, our prototype on Firefox 8.0 contains all the interfaces discussed in this paper, including the support required for our case studies. In Section 5.3.3, we will show that with this set of hooks, BSM supports the functionality required by existing browser-based security solutions. In Section 5.3.5, we demonstrate BSM’s potential in supporting new applications.

The sample security module in the Firefox 8.0 has an optimized implementation of the Event Tracker and Browser Object Access Helper. It also contains a reference implementation of the solution proposed in ESCUDO written in less than 50 SLOC.

For the communication between the instrumented browser and security modules, we added a dynamic library to Firefox, which was linked with all of Firefox’s own dynamic libraries during the building of Firefox source. This new dynamic library serves as the bridge between the browser and security modules, which in turn used `dlopen` to load security modules. BSM Hooks invoke security modules via function pointers stored in the connector. With such an implementation, security modules are independent of the instrumented browser, and end users can switch between different security modules without the need of rebuilding the instrumented browser.

5.3.2 Supporting Security Requirements

Intercepting important operations Table 5.3 summarizes the mapping between interfaces of different categories in our reference model and sample files where we inserted hooks in our prototype on the three browsers.

Category	Firefox 3.5	Firefox 8.0
1 Network Connection	nsHttpChannel.cpp	As in Firefox 3.5
2 Network->Parser	nsScanner.cpp	nsHtml5StreamParser.cpp
3 Parser->DOM	nsParser.cpp, CNavDTD.cpp	nsHtml5StreamParser.cpp
4 DOM->Parser	nsHTMLContentSink.cpp	nsHTMLContentSink.cpp
5 DOM->Network	nsGenericElement.cpp, nsScriptLoader.cpp, nsFrameLoader.cpp, nsImageLoadingContent.cpp	As in Firefox 3.5, and nsContentUtils.cpp
6 DOM->JS	nsScriptLoader.cpp	As in Firefox 3.5
7 JS->DOM	nsDocument.cpp, nsHTMLDocument.cpp, nsLocation.cpp, nsGlobalWindow.cpp, xpcwrappednative.cpp, xpcconvert.cpp, qsgen.py	As in Firefox 3.5
8 DOM->Event	nsGenericElement.cpp	As in Firefox 3.5
9 JS->Event	nsGlobalWindow.cpp, nsDocument.cpp, nsGenericElement.cpp	As in Firefox 3.5
10 Event->JS	nsJSEnvironment.cpp, nsXMLHttpRequest.cpp	As in Firefox 3.5
11 JS->Network	nsXMLHttpRequest.cpp	As in Firefox 3.5
12 Event->Network	nsHttpChannel.cpp	As in Firefox 3.5
15 Event Processing	nsWindow.cpp, nsWebShell.cpp, nsEventListenerManager.cpp	As in Firefox 3.5
17 Add-ons	nsPluginHostImpl.cpp	nsNPAPIPlugin.cpp, nsPluginHost.cpp, nsPluginInstanceOwner.cpp

Table 5.3: BSM Implementation in Different Browsers

Supporting fine-grained principals In our prototype in Firefox 3.5 and 8.0, to support fine-grained principals, we used an additional map to record the principals of scripts on the page. When a script is first processed, we record its principal in the map, and when later the script executes, we retrieve its principal from the map. Our prototype also supports propagation of principals with the hook to create a new script element.

Maintaining context information We implemented the Event Tracker in Firefox 3.5 and 8.0. Each time when a BSM Hook at the entry of an intercepted function is invoked, we push an entry to our Event Tracker stack, which contains the information of the function as well as important arguments. When a BSM Hook before the exit of an intercepted function is invoked, we pop its entry from our Event Tracker stack accordingly. This Event Tracker stack is available to security modules to examine for the callers of the current function.

Supporting access to browser states The Browser Object Access Helper is implemented as a separate dynamic library that enables security modules to access Firefox data types through its helper functions. Currently, we have implemented several example helper functions for accessing HTML element attributes as well as conversions between custom and Firefox-specific types to demonstrate its functionality. Security module developers can add more to support their own requirements.

5.3.3 Case Studies: Supporting Existing Security Solutions

MashupOS MashupOS [168] proposes a few new primitives for including third-party web content into the integrator pages, aiming at solving the all-or-nothing (i.e., `<script>`

or `<iframe>`) dilemma that currently exist. For *unauthorized content*, where the integrator does not trust the third-party content embedded, two types of sandbox primitives are proposed to host different type of unauthorized content. Both of the two types do not have any built-in principal, so they cannot access any other web content except their enclosing sandboxes.

- `<Sandbox>` For private unauthorized content. Only the integrator from the same origin can access the content inside a `<Sandbox>`.
- `<OpenSandbox>` For open access, i.e., any principal has direct access to content embedded in it.

Another two primitives are also proposed for access-controlled content: `<ServiceInstance>` and `<Friv>`. The `<ServiceInstance>` abstraction provides an isolated environment for hosting third-party content. The `<ServiceInstance>` abstraction does not have display resource, and the parent of `<ServiceInstance>` primitives can assign `<Friv>` abstractions to them to provide display.

Communications between the primitives above are supported by `CommRequest` and `CommServer`, two new objects exposed to web application JavaScript.

Now we will show how BSM supports the implementation of the different functionalities required by MashupOS.

New primitives As all the proposed new primitives were actually implemented using iframes in MashupOS, and access controls were performed by implementing DOM object wrappers, they require intercepting sensitive operations and events and extracting context information. With support from BSM, new access control rules with the proposed primitives can be implemented with the BSM hook to `js_dom_call`, as illustrated in Figure 5.2.

Here we first obtain the reference to the subject from the Event Tracker that records the dependency relationship between triggering events and the subsequent actions. Such relationship is built up according to control and/or data dependency. Then we get the origins and names of the subject and the object passed to the hook from the browser function. Finally, we apply the checks according to the specifications of the new primitives.

Alternatively, access control specific to a particular scriptable DOM interface can also be implemented separately in the corresponding hook. For example, we may impose specific restriction to `document.getElementById` as shown in Figure 5.3.

In the original implementation of MashupOS, `<OpenSandbox>` was not implemented. This might be because it would actually need to *relax* the existing same-origin policy in current browsers, if implemented based on iframes. Our BSM framework supports flexible and finer-grained principals to be defined by security modules. With the hook to the DOM access checker, new access control rules proposed for

```

int js_dom_call_hook(Object * object, ...) {
    ...
    Object * subject = get_subject_from_dstack();
    char * subject_origin = get_origin(subject);
    char * object_origin = get_origin(object);
    char * subject_name = get_name(subject);
    char * object_name = get_name(object);

    if (subject_name == NAME_SANDBOX) {
        if (object_name != NAME_SANDBOX) {
            return DENY;
        }
    }
    else if (object_name == NAME_SANDBOX) {
        if (!subject.is_ancestor_of(object)) {
            return DENY;
        }
    }
    else if (object_name == NAME_OPENSANDBOX) {
        return ALLOW;
    }
    else if (subject_origin == object_origin) {
        return ALLOW;
    }
    else {
        return DENY;
    }
}

```

Figure 5.2: MashupOS: New Primitives

```

int document_getelementbyid_hook(Document * doc, char * id) {
    Object * subject = get_subject_from_dstack();
    char * subject_url = get_url(subject);
    if (contains(safe_url_list, subject_url)) {
        return ALLOW;
    }
    else {
        return DENY;
    }
}

```

Figure 5.3: MashupOS: Specific Access Control Example

<OpenSandbox> can be implemented in a similar way as other proposed primitives, as shown in Figure 5.4.

This example illustrates the *extensibility* BSM provides to security modules, as they are allowed to redefine existing principals and access control rules in browsers. However, great caution must be taken by security modules when changing the existing security principals and mechanisms, and they are responsible for verifying the newly implemented security model will not weaken existing security guarantees.

CommRequest To implement CommRequest proposed by MashupOS, two new host objects need to be constructed and exposed to JavaScript. This can be implemented in JavaScript with the `postMessage` cross-origin communication mechanism, illustrated in Figure 5.5.

Shown in Figure 5.6, this piece of JavaScript needs to be inserted into the JavaScript contexts when they are initialized, with the hook that intercepts the execution of JavaScript.


```

int check_dom_property_access_hook(char * framename) {
    if (framename != NULL &&
        framename == NAME_OPENSANDBOX) {
        return SKIP_SOP;
    }
    else {
        return SOP;
    }
}

```

Figure 5.4: MashupOS: <OpenSandbox>

```

// JavaScript
var url_to_window_map = new Array();
var CommServer = new Object();
CommServer.listenTo =
    function(port, handler) {
        url_to_window_map['`local:`` + location.href + port] = window;
        window.addEventListener(`commreq`, handler, false);
    }
var CommRequest = new Object();
CommRequest.def_handler = function(req) {
    if (req.origin === location.origin) {
        this.responseBody = req.data;
    }
}
CommRequest.open = function(action, url) {
    this.win = url_to_window_map[url];
    this.url = url;
    window.addEventListener(`commreq`, this.def_handler, false);
}
CommRequest.send = function(data) {
    postMessage(data, url);
}

```

Figure 5.5: MashupOS: CommRequest in JavaScript

```

int js_eval_script_hook(out string script_to_insert) {
    script_to_insert = "...scripts to execute...";
    ...
}

```

Figure 5.6: MashupOS: JavaScript Insertion

```

int parser_read_buffer_hook(out string buffer, out int buf_len) {
    search_list[0] = ``sandbox``;
    replace_list[0] = ``iframe``;
    search_list[1] = ``opensandbox``;
    replace_list[1] = ``iframe``;
    search_list[2] = ``serviceinstance``;
    replace_list[2] = ``iframe``;

    for (i = 0; i < search_list.length(); i++) {
        string search_tag = search_list[i];
        string replace_tag = replace_list[i];
        string search_opening = ``<`` + search_tag;
        string search_end = ``</`` + search_tag + ``>``;
        int opening_index = buffer.search(search_opening);
        int end_index = buffer.search(search_end);
        while (opening_index) {
            string orig_tag = buffer.substr(opening_index, end_index);
            buffer.replace(opening_index, search_opening.length(),
                ``<script><!--/* `` + orig_tag +
                `` */--></script>`` + replace_tag);
            end_index = buffer.search(search_end);
            buffer.replace(end_index, search_end.length(),
                ``</`` + replace_tag + ``>``);
            opening_index = buffer.search(end_index + 1,
                search_opening);
            end_index = buffer.search(end_index + 1,
                search_end);
        }
    }

    buf_len = buffer.length();
    ...
}

```

Figure 5.7: MashupOS: Primitive Translation

Translation from new primitives to existing tags As all new primitives need to be transformed into iframes before they can be processed by web browsers, MashupOS needs to intercept HTTP responses and translate all new primitives into iframes while appending some identification information so that MashupOS DOM object wrapper would identify the underlying primitives denoted by the iframes. This is supported in our framework as follows. After the hook to the parser’s buffer reader of the HTTP response is invoked, it starts to match the opening and closing tags in the search list, replaces them with iframes, and appends their original content into script comments for future processing by DOM object wrappers for access control. Figure 5.7 illustrates such implementation.

Other browser security solutions In the rest of this section, we briefly discuss how functionalities in other security solutions are supported by BSM.

ESCUDO [82] adopts a ring-based security model to perform access control on host objects in web applications. Their basic idea is to further partition web application content into different regions with different ring numbers, and each region can specify regions with which ring numbers can access it for reading, writing, or implicit access. It requires interception to JavaScript-to-DOM calls and defining finer-grained principals, which are supported by BSM Hooks `js_dom_call_hook`, and the `get_principal`, `set_principal` pair, respectively. We have implemented a ref-

erence implementation in the sample security module of our prototype, with less than 50 SLOC.

Similar to MashupOS, **OMash** [37] also offers fine-grained access control for mashup applications, while differing in 1) providing a single abstraction for expressing all trust relationships, and 2) independent of SOP. Its implementation requires intercepting JavaScript's accesses to DOM and authentication credentials. BSM supports it with the hook to JavaScript-to-DOM calls and the hook to the `document_cookie` interface, respectively,

However, in their actual implementation to mediate DOM access in a Firefox extension by setting Configurable Security Policies (CAPS), SOP was still retained as it would require browser modification, otherwise. BSM supports all its required functionalities without the restriction of the SOP currently in the browser.

Besides, for the mediation of authentication credentials, their original implementation needed to modify 3 lines of Firefox code regarding access to cookies. This is readily supported by our hook to cookie setter and getter functions, `document_cookie_hook`.

App Isolation [31] analyzes the security benefit of using separate web browsers for sensitive and non-sensitive online transactions, and summarizes it into two aspects: entry-point restriction and state isolation, which are supported by the interception of JavaScript to DOM access with the hook `httpchannel_init_hook`, and with the hooks to persistent storage data, such as `cookie_obj_access_hook` and `localStorage` `localStorage_getitem`, etc. Our current prototype implementation of BSM does not support process-level isolation as implemented in App Isolation.

The entry-point restriction restricts requests to entry-point URLs from a different web application. With the hook `httpchannel_init_hook` to the initialization of a new HTTP channel, entry-point restriction can apply checks on which URLs are allowed to load from.

The other aspect is state isolation, including the isolation of in-memory state and persistent state. Our current prototype implementation of BSM is based on Firefox, and does not support process-level isolation. However, BSM provides the functionality required to achieve the same effect of persistent state isolation with the hooks to cookies `cookie_obj_access_hook` and `localStorage` `localStorage_getitem`, etc. With these hooks, we can overwrite the existing cookie access logic in the web browser, and return a different cookie or `localStorage` object according to different applications.

Noncespaces [67] proposes an approach to assign randomized prefixes to HTML tags at the server side to detect injected script tags that would not have the current prefixes.

The client-side implementation is to fetch and enforce the policy on the received HTML content, which are supported by the interception of HTTP response processing using hooks `process_response_hook` and the interception of parsing with `parser_read_buffer_hook`, respectively, as shown in Figure 5.8.

The policy specifies which untrusted tags are allowed for processing. The policy

and the random prefix are transmitted to the client side in HTTP headers. The original solution was implemented in a proxy, but it can also be readily implemented using our framework. With the support of BSM, the policy and prefix information can be retrieved using the hook to the function processing HTTP responses `process_response_hook`, and policy enforcement can be implemented in the hook to the parser's reader of the HTTP response buffer `parser_read_buffer_hook`, where an external parser is employed to enforce security rules.

```
int process_response_hook(array_list headers, array_list values) {
    for (i = 0; i < headers.length(); i++) {
        if (headers[i] == HEADER_NONSPACE_POLICY) {
            nonspace_policy = values[i];
        }
        else if (headers[i] == HEADER_NONSPACE_PREFIX) {
            nonspace_prefix = values[i];
        }
    }
    ...
}

int parser_read_buffer_hook(out string buffer, out int buf_len) {
    // init an external parser
    DTDSpec * d_spec = new DTDSpec(filename);
    XMLParser * parser = new XMLParser(buffer, d_spec);
    try {
        parser->parse();
    }
    catch(XMLException e) {
        ...
        return DENY;
    }
    ...
}
```

Figure 5.8: Noncespaces: Policy Enforcement and Prefix Parsing

Noxes [88] is a lightweight firewall-like application for web-based attacks. With user-specified or inferred rules, it can detect some XSS and privacy violation attacks. Its original implementation was not integrated into web browsers. BSM supports the link extraction with the interception to the operation of the parser reading the HTTP response buffer, via `parser_read_buffer_hook`, and to the HTTP request initialization, via `http_channel_init_hook`. Noxes also inserts a piece of “controlling” JavaScript code in the beginning of each web page to control the pop-up windows and frames, and a warning will be shown to the user asking whether to continue it or not. With our framework, it can be implemented in a non-invasive manner with the interception to `window_open` and `frameloader_load` operations.

Jim et al. [83] propose a mechanism for web applications to specify security policies that are enforced in web browsers. Before any script is executed, a JavaScript security hook provided by the web application is invoked. The browser proceeds to execute the script only if the security hook returns true. The hook proposed in this work corresponds to `scriptloader_process_script_hook` in BSM.

Another work by Hallaraker et al. [68] mainly consists of two parts. The first is an auditing module that records JavaScript's access to host objects, and the second part is a simple misuse IDS. Its instrumentation to the web browser is supported by BSM's

Test Suite	Original Firefox (runs/s)	BSM with Lib (runs/s)	BSM w/o Lib (runs/s)
Recommended	287.76	245.5 (17.21%)	284.13 (1.27%)
All DOM	1286.41	1144.39 (12.41%)	1269.77 (1.31%)
JS test	165.49	163.19 (1.40%)	164.5 (0.60%)

Table 5.4: Dromaeo Test Results for BSM

interception to JS-to-DOM calls with `js_dom_call_hook`.

5.3.4 Performance

We evaluated the performance impact introduced by our prototype of BSM in Firefox 8.0². All the experiments were conducted on a computer with Intel Core 2 Duo CPU at 2.33GHZ, 250GB 7200 rpm disk, and 4GB RAM, running Ubuntu 11.10.

We measured the performance overhead in two ways. One was the *page load time*, which is defined as the time between HTTP request initialization and the *page onload* event. The other evaluation method was using *vendor benchmarks* to measure the impact on the JavaScript execution. We compared performance between the Firefox with BSM prototype and the unmodified Firefox.

Page load time To assess impact of our approach on user browsing experience, we measured the page load time. The *page load time* represents how long it takes to load a page. We used Alexa top 10 websites (in early 2012) to measured page overhead. The delay in page load time is negligible³.

The page load time does not give an accurate assessment on the actual performance overhead incurred by our prototype, as other factors dominate the load time. However, this experiment demonstrated that incorporating BSM into Firefox 8.0 would not cause user experience degradation even with a security module of a reference implementation of ESCUDO [82].

Vendor benchmarks We also evaluated the performance with vendor benchmarks provided by Dromaeo [110]. Dromaeo is a benchmark suite provided by Mozilla to test web browsers' performance of JavaScript and DOM. Table 5.4 shows the experiment results of the *recommended*, *all DOM* and *JS test* tests of Dromaeo. We observed a 1.27% overhead on the *recommended* test suite, a 1.31% overhead on *all DOM* test suite and a 0.60% overhead on *JS test* without any security module enabled. With Event Tracker and Browser Object Access Helper, and a security module of a reference ESCUDO implementation, we noticed a 17.21% overhead on the *recommended* test suite, a 12.41% overhead on *all DOM* test suite and a 1.40% overhead on *JS test*. The *all DOM* and *recommended* test suites incurred larger overhead, because the security module needs to check for the `ring` attribute for each DOM access from JavaScript, and the benchmark contains focused and repetitive DOM accesses. We can see that

²We did not measure the performance overhead on our prototype with Firefox 3.5 since that is an experimental platform with many hooks currently not leveraged to support any security module.

³In fact, during our experiments of page load time averaged over 10 runs, we were not able to get stable results showing that it would take more time to load a web page with BSM than in the original Firefox.

```

//A sample web page from www.example.com

<iframe
  src="http://www.thirdparty.com/page.html" />
<a href="#" id='link1'>click here</a>
<script>
// modify link
var lk=document.getElementById('link1');
lk.href="www.example.com/page2.htm";
//Create new script element that sends XHR
//request to example.com
....
</script>

```

Figure 5.9: A Sample Web Page Snippet

BSM itself only incurs little performance overhead, while security solutions developed with BSM would incur reasonable overhead.

5.3.5 Supporting New Security Applications

Extracting request dependency to detect session misuse attacks Web requests are the cornerstone of modern web applications. Web application servers expose HTTP-based interfaces to browser-based web clients, allowing them to request services or resources from servers. In the increasingly complex browser environment, web requests can be generated in many ways, which is actively explored by attackers to attack web applications. In contrast, web servers only have limited information to decide whether a request is generated legitimately or triggered by attackers. As a result, many attack techniques have been developed to hijack or ride users' web sessions through malicious web requests, including cross-site scripting (XSS) [120], cross-site request forgery (CSRF) [14], and clickjacking [70], which we call *session misuse attacks*. Recently, attackers turned to a new class of attacks, which use social engineering to trick users to assist session misuse attacks [46, 26].

With the support provided by BSM, we extract request dependency at run time to detect anomalous requests. For example, let us consider a sample web page snippet from *www.example.com* that initially contains a link, an image, an iframe and a script element. The script modifies the attribute of the link and also creates another piece of script on the fly, which in-turn sends XMLHttpRequest request. The iframe loads a page from a third-party server, which in turn loads an image and a script element. This is shown in Figure 5.9.

Figure 5.10 shows the request dependency extracted for this page. As shown in the figure, *www.example.com* is a web domain that generates various HTTP requests. First, it sends HttpRequest0 for its front page. Then during parsing, it sends HttpRequest1 and HttpRequest2 to load an iframe and an image resource, respectively. The iframe further requests HttpRequest4 and HttpRequest5 for external image and script resources, and the script in the iframe also generates a new request HttpRequest6 during its execution.

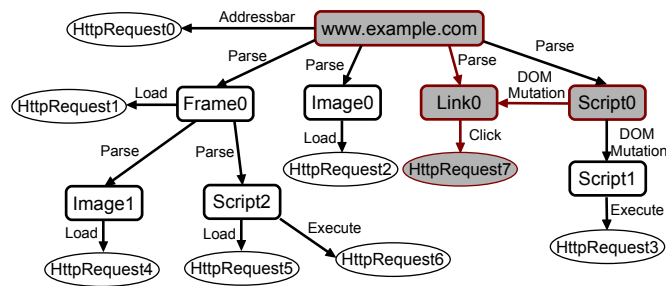


Figure 5.10: Graph Showing Request Dependency for The Sample Web Page Snippet

Back to the main page, some inline script creates a new script element by mutating the DOM, which in turn sends out `HttpRequest7`. The inline script `Script0` also modifies the HTML hyperlink. The link is clicked, so it sends `HttpRequest7` to navigate to another page. We further *slice* the RDG for specific requests to reduce the size of data communicated to servers. In Figure 5.10, the slice extracted for `HttpRequest7` is shown in gray color. With such request dependency, we can detect malicious requests that are generated in an unexpected way. For example, if during run time our system detects `HttpRequest7` is generated by a piece of JavaScript, rather than by a user click, it is alerted since the way of generating the request is suspicious.

Leveraging the security support provided by BSM, we are able to detect existing and new session misuse attacks with request dependency. This application demonstrates the potential of the general security support of BSM in supporting new security applications to solve new security issues.

Capturing evolving client-side states of web applications We can also leverage BSM to address challenges in diagnosing foreign behaviors in web applications [44]. With Ajax and HTML5 technologies, a large fraction of such behaviors occur at the client side. Since HTML5 and its supporting frameworks [3, 74] introduce new APIs to mobile applications, these new type of applications have access to more sensitive data to traditional web applications on PCs. For example, HTML5-based mobile applications may have direct access to camera, geolocation, contacts and files on users' mobile devices, more security and privacy concerns will arise. Since these applications may also dynamically take in web contents from third-parties for execution, even legitimate applications are susceptible to web-based attacks.

Figure 5.11 illustrates a sample fragment of a normal behavior model for Twitter. The model is extracted by a tool built on top of BSM. By registering to BSM hooks, it monitors the behaviors of web applications and establishes the correlation between different events and operations.

From the model, we can see that, for example, a normal procedure of posting a tweet would be to first login, and then to click the “Tweet” button. Such a model can be used to diagnose abnormal behaviors in XSS attacks, e.g., sending a tweet posting request on a page rather than “Timeline”, or automatically generating the posting request by JavaScript without users' consent.

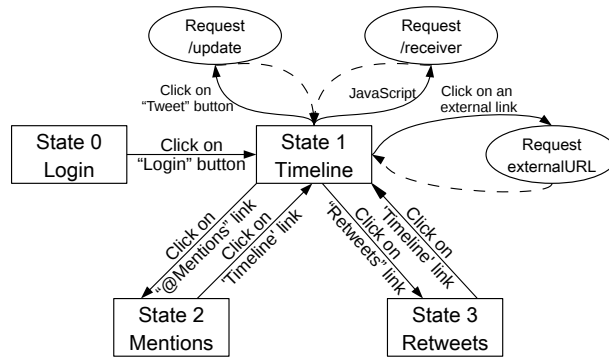


Figure 5.11: A Sample Behavior Model

Rectangles denote Application States, and ovals denote Action States. Transitions between states are marked with triggering events.

5.4 Discussion and Future Work

Completeness To verify the feasibility of the BSM design, we have implemented a prototype on Firefox. We are working on migrating BSM to the Android browser and Chromium, both of which are based on WebKit.

At this stage, our prototype has no guarantee on the completeness in mediating sensitive operations in the browser. We understand that given the complexity of modern browser implementations, proving the completeness of the entire BSM is prohibitively difficult, and we are planning to start with partial verification on the completeness of BSM on the JavaScript engine, using program analysis techniques such as those in [185, 48].

Flexible access control with a high-level policy language Currently, security modules of BSM have to be written in C/C++. Although this provides the greatest flexibility to develop security solutions, it is not convenient and not entirely portable on different operating systems. As our ongoing work, we are designing a flexible access control framework to allow security solutions to regulate web application behaviors with a high-level policy language, without the need of writing C/C++ code. This access control framework will allow the specification of conditional and contextual transition of principals, enabling adaptive policy enforcement with finer-grained control under different scenarios. For example, in the beginning an untrusted script may have full privilege as the hosting page until suspicious behaviors are detected out it, such as accessing the user's account number or password on the page, when its privilege is degraded according to the policy.

Supporting information flow tracking in JavaScript Our current prototype does not support information flow tracking inside the JavaScript engine, and we are going to include it in our future implementation to provide additional support.

Regulating browser add-on behaviors Browser add-ons can run binary code and directly communicate with underlying operating system as well as any component in our reference model. Therefore, they are out of the scope of this work. To capture and regulate their behaviors, traditional static and dynamic approaches on binary programs are required.

5.5 Related Work

In this section, we briefly discuss related work in web security as well as traditional security research works that share similar ideas with our work.

5.5.1 New Web Browser Architecture

Recently, security researchers have proposed several new browser architectures as a more secure runtime environment for web applications. The main idea in this line of work is to use OS mechanisms to isolate components or resources in the browser, and mediate the access among different components or resources. The detailed methodology in isolation varies with different approaches. For example, Tahoma [36] uses virtual machines to isolate web applications. The OP browser [62] isolates different browser components, called subsystems, i.e., the browser kernel, storage, network, UI and web page instances using processes. The Chromium browser [15, 136] and Gazelle [169] also use processes to isolate rendering engines for web pages, according to site instance, and origins, respectively. Tang et al. [154] reduces the trusted computing base by designing an operating system specifically for browsers. These research works are complementary to BSM as they mitigate attacks against browsers' memory vulnerabilities, rather than those on web applications' integrity and user data.

Mickens et al. propose Atlantis [106], which is a new browser design that provides low-level APIs for web applications to customize their execution environment. The primary goal of Atlantis is to enable more control from web applications over the complex, obscure and potentially incompatible implementation of different browsers. Compared to Atlantis, BSM is designed specifically for enhancing security support in web browsers, and such support is provided to security solution developers, rather than to web applications.

5.5.2 Defenses to Attacks on Web Applications

On the other hand, various solutions have been proposed to address the threats to web applications.

For client-side solutions, Yue et al. [182] instrument web browsers to obtain JavaScript execution traces and perform offline analysis on these traces. This is similar as the work of Hallaraker et al. [68], which audits JavaScript code executions and then analyze the traces using misuse IDS, that is, based on known signatures. They need to intercept the interfaces between the JavaScript engine and DOM. Alhambra [153] proposes a

browser-based system that enables automatic creation, enforcement as well as testing of security policies to prevent web-based attacks without the need of modifying web server or web applications. Compared to it, BSM is a more general framework that provides more comprehensive interception of browser operations and events, extensible functionality as well as deep diagnosis of web application behaviors. However, the current prototype of BSM does not support taint tracking, which is supported by Alhambra.

Information flow tracking is an important technique that has a wide range of applications in web security, including detecting malicious behaviors in browser extensions [41], preventing XSS attacks from leaking sensitive session information [163], and enhancing policy enforcement on a wide range of attacks [177], etc. As our current BSM framework works at the level of major browser components, our prototype does not support it. As discussed in Section 5.4, we will include support for it in future development.

In previous sections, we have also discussed other related works for mashup protection and XSS defense [158, 88, 168, 37, 82, 67, 83, 65, 152]. Their need of modifying the browser can be supported by BSM.

More recently, researchers start to collaborate the server and the client side together to detect and prevent web-based attacks. One example is BLUEPRINT by Ter Louw et al. [159]. It replaces the attack-prone document parsing process in web browsers with a parse tree generated on web servers. Another example is Document Structure Integrity (DSI) by Nadji et al. [114] that proposes a solution involving both the server-side and the client-side to enforce the document structure integrity and prevents untrusted content from compromising the trusted data. They require interception to the parsing process and JavaScript taint tracking. BSM provides support to the parsing interception.

5.5.3 Operating System Security

Solutions on access control [143, 19, 9] have been proposed for operating system security, and the most closely related work to ours is Linux Security Modules (LSM) [176]. LSM is a general framework that supports various access control enhancements to the original discretionary access controls in Linux. It provides security mechanisms hooks to important Linux kernel functions to allow them to enforce access control policies.

5.6 Summary

In this work, we analyze the weaknesses of current security mechanisms in web browsers, and identify the key security support required to protect web applications. We propose Browser Security Modules (BSM) to provide general security support for web applications by intercepting sensitive events and operations in the browser and maintaining correlations among them. Through extensible interfaces, BSM allows easier development of security solutions and experiments with new browser security mechanisms. We

prototyped BSM in Firefox, and demonstrated its functionality by demonstrating its support for existing and new browser-side security solutions.

Chapter 6

Protecting Sensitive Web Content from Client-side Vulnerabilities with CRYPTONS

Presently, web browsers are designed to isolate content between web origins [113], preventing data owned by one origin from being accessed by other origins. However, in practice, information processed within a web origin’s protection domain often has different levels of sensitivity. Some application data — which we refer to as *sensitive data* in this work — is critically sensitive and more important than the rest of application data. Social security numbers, medical reports, enterprise emails with financial data, tax-related information, bank transactions, and user passwords are all examples of such sensitive data. Web application owners often wish to strongly isolate sensitive data within their applications, to protect it from vulnerabilities resulting from processing non-sensitive information in its code.

Under the origin-based model, several applications processing sensitive data, such as password managers [91] and encrypted chat or email clients [89], have to completely trust the client-side browser’s origin isolation abstractions and the correctness of its client-side application code [5, 150]. Information belonging to one origin is accessible to all application code running within the protection domain of that origin, as well as browser code (including add-ons), even if they do not need to access such information. The client-side web application and the underlying browsers consist of millions of lines of code and have had their steady share of security vulnerabilities historically; client-side application vulnerabilities (like DOM-XSS [145]) are pervasive in client-side application code and browser addons [28]. Given the scope of client-side vulnerabilities, protecting sensitive data within the browser environment is a serious practical concern.

Problem. The crux of the problem is that the web lacks abstractions for information owners to specify what information is sensitive and how it must be processed in the client-side web browser environment. To address this problem, we envision new abstractions that give owners control over the processing during the lifetime of selective

sensitive data in the client-side browser. We term these as *intra-origin data control* abstractions. In particular, we advocate that web browsers provide these controls as *first-class* abstractions on the web, independently of strengthening the browser’s origin isolation.

Although there has been extensive prior work on browser design, principled mechanisms for intra-origin information control have not received much direct attention. For instance, recent browser designs have significantly improved privilege separation between code components [36, 15, 63, 169, 154, 31, 158, 13]. However, these techniques do not directly allow web applications to enforce control on their data; a compromised component with access to sensitive data can still launder it through its export interfaces. For example, the sandboxed renderer process in Google Chrome [15], if compromised, can still exploit legitimate network interfaces provided by the Chrome kernel to leak data to attack-controlled domains (say via an `img` load) [32]. Although information control has been investigated through language-based enforcements in specific languages [105, 150, 72, 37], these mechanisms fail to extend the protection to the lowest level (e.g. against attacks targeting low-level memory access). Although piecemeal techniques (such as SFI [167], JIT hardening [140], CSP [150], and language-based techniques) could be a plausible basis for sensitive data protection, we argue to build more direct information control primitives on the present web instead.

Our Solution. We design and implement a novel data abstraction and browser primitive CRYPTON that enables protecting sensitive data throughout their lifetime in the browser. CRYPTONS are programmed directly in existing web languages (HTML/JavaScript), without requiring major re-engineering of existing application logic. A CRYPTON explicitly marks a unit of sensitive data in a web page; its semantics enforce that sensitive data and information computed from them are tightly isolated at the lowest level (in the raw program memory). To support rich web applications that need to execute flexible operations on sensitive data, CRYPTONS tie sensitive data with functions that are allowed to compute over them. These few select CRYPTON functions are trusted to be statically verified by web developers and are the only channels for releasing information about sensitive web content. In addition, a CRYPTON provides other important capabilities for sensitive data in web applications — guaranteed rendering (with a proof-of-impression), and certified delivery of user inputs in the browser.

Building trustworthy implementations of these abstractions is an important, but challenging goal. The main challenge stems from the *monolithic* trust model of today’s web — application servers have to completely trust the client-side code in web browsers, add-ons and applications.

To address this practical concern, we rethink the monolithic trust model that web applications have on client-side code. In our design, the web server trusts a small piece of client-side code called the CRYPTON-KERNEL— a small, trusted standalone engine, which acts as a root-of-trust for the server on the client device. CRYPTON-KERNEL runs in a separate OS process and sandboxes the untrusted browser, which invokes

the CRYPTON-KERNEL on-demand to securely interpret CRYPTONS. A compromised browser can deny access to sensitive information (denial-of-service) by refusing to invoke the CRYPTON-KERNEL, but it cannot subvert the integrity and confidentiality of sensitive data. In effect, our design of CRYPTON-KERNEL allows web applications to specify information controls on sensitive information, bootstrap rich computation on it and to build trusted paths between the user and the sensitive application code [118].

Unlike prior research on origin isolation in browsers, we do not isolate code components or memory regions; instead we isolate sensitive data within a web application’s origin using authenticated encryption. CRYPTONS make novel use of the concept of *memory encryption*[95], wherein data computed from sensitive data remain encrypted throughout their lifetime inside the untrusted browser’s memory. This ensures that any unintended malicious or compromised code, whether part of the browser or the application logic, can only access sensitive data in their encrypted form. Memory encryption allows sensitive data to be opaquely accessed by an untrusted browser functionality (such as the network stack, data stores, language parsers, rendering logic, and so on) as well as application logic, without risking their confidentiality and integrity.

Therefore, the CRYPTON-KERNEL enforces its security guarantees while reusing browser functionality — it does not rely on the browser’s same-origin policy enforcement, its components such as the HTML parser, the JavaScript/CSS engine, and the network engine to ensure its security properties. CRYPTON-KERNEL has a TCB about 40 – 50 times smaller than that of a real web browser, such as Firefox and Google Chrome, consisting of roughly 20K lines of code. Through manual analysis, we find that this design can prevent more than 92.5% of known security vulnerabilities from exfiltrating or tampering with sensitive information, in a web browser such as Firefox (Section 6.3).

Evaluation on Real-world Applications. First, we manually modify 3 popular web applications to selectively use CRYPTONS to protect sensitive web content within them, including a webmail, a web messenger and a web forum. The results show that CRYPTON-KERNEL strengthens their security against a broad range of client-side vulnerabilities, with a small (~3-5 days) of adoption effort. Next, we conduct an extensive macro evaluation of applicability of CRYPTONS on real-world web sites including Alexa Top 50 websites, including account signup pages like Gmail, social networking sites like Facebook, banking sites like Bank of America, e-commerce applications like eBay, and several others. We treat all user inputs (form fields, text-boxes, selections, etc.) and other manually marked contents in these case studies as sensitive information. Our macro study shows the effectiveness of our solution and small adoption cost in these real-world web applications. Finally, we evaluate the performance of our initial prototype. Although our WebKit-based prototype implementation is still an unoptimized proof-of-concept, we do not perceive noticeable slowdown when running it over real-world applications. Further evaluation also shows that performance overhead from encryption/decryption operations is also modest, less than 8% when 100% of the texts on 5

test pages from the Dromaeo web benchmarks are marked as sensitive.

Contributions. In summary, we make the following contributions in this work.

- We propose a novel solution for protecting sensitive web content with a small trusted computing base (TCB), roughly $40x - 50x$ smaller than a full browser. Our design advocates rethinking the monolithic (all-or-nothing) trust model between servers and clients on the existing web.
- We employ memory encryption to protect sensitive data in web browsers at a byte-level granularity, offering strong protection at a low overhead. To the best of our knowledge, we are the first to propose a data-centric abstraction for end-to-end data protection in web browsers.
- We perform a large-scale evaluation of the applicability of intra-origin abstractions like CRYPTON in existing popular applications. We demonstrate the adoptability of these abstractions into a large number of real-world web sites today with a small developer effort.

6.1 Problem & Overview

To illustrate the need for new abstractions for sensitive data, consider the example of a webmail application, such as Gmail. Currently, all emails rendered to the user by the client-side Gmail application are treated equally in protection. However, this does not match the security requirements in practical settings. Users may use the Gmail account to access both leisure emails and corporate emails. As corporate emails may contain sensitive information pertaining to trade secrets or financial information, for corporate security, a user Alice may be willing to instruct Gmail to mark her corporate emails as sensitive and strongly protect them from being leaked or tampered with. Unfortunately, even if Alice and her webmail service provider (Gmail) are willing, it is difficult for Gmail to guarantee the privacy of sensitive emails when they are processed and displayed in the client’s web browser. The current web provides no abstractions for Gmail to isolate enterprise emails and specify controlled (but rich) computation on it at the client-side.

6.1.1 Threat Model

In web applications like the above Gmail example, sensitive data are exposed to a large threat landscape due to vulnerabilities in the client-side web stack. We give a back-of-the-envelope enumeration of the in-scope threats of our approach, based on our investigation of vulnerability databases [55].

- *Browser components.* Vulnerabilities in browser components can lead to violation of the confidentiality and integrity of sensitive information. We measure the number of such vulnerabilities in Mozilla Firefox. We manually study security

vulnerabilities of Firefox over the past 7 years from its bug database [55]. We find that out of the 360 vulnerabilities accessible to us, at least 280 (80%) ones could expose sensitive information to arbitrary malicious script or binary code running in the browser’s chrome privilege. We present more details of our methodology in Section 6.3.4.

- *Browser add-ons.* Browser add-ons and extensions may access sensitive data processed in browsers. Although web browsers may prompt users to approve the permissions requested by add-ons during their install time, many users may “click-through” to grant them to the addons. We surveyed the Top 30 Chrome extensions, and find that 23 request access to Gmail; if these are installed and have vulnerabilities, they can be compromised by attackers to access sensitive email. 15 out these 23 have been confirmed to contain code-injection vulnerabilities in Carlini et al.’s recent work [28].
- *Application components.* Web application components can also access the sensitive data in web applications. The Gmail Inbox page contains 171.1 KB code (including HTML, CSS, JS), while only 1.7 KB of it needs to access email contents. However, client-side web application vulnerabilities are pervasive, including mixed content, unsafe `evals`, capability leaks, DOM-XSS, and insufficient origin validation.¹

In this work, we focus on providing two security guarantees on sensitive data processed in the client-side browser: *integrity* and *confidentiality*. Our design assumes that there are client-side vulnerabilities in web applications (including add-ons) and browsers, but trusts the underlying OS, its windowing and graphics interfaces. Our defenses preclude many vulnerabilities (outlined above) from corrupting sensitive data or leaking unprotected sensitive data to adversaries. For instance, attackers can exploit client-side vulnerabilities to leak sensitive emails to remote adversaries. However, those emails are in encrypted form and the decryption keys are inaccessible to attackers. Several attacks are outside the scope of the guaranteed security properties; we discuss them in Section 6.1.4.

6.1.2 CRYPTON: A New Abstraction

We introduce a new data protection abstraction called CRYPTON. Conceptually, CRYPTONS are akin to classes in object-oriented languages, which encapsulate sensitive data together with the operations that can legitimately operate on them. CRYPTONS provide the following 4 main capabilities.

Information isolation. We enable transparent isolation of sensitive data from other data and code in web browsers. In our Gmail example discussed earlier, the confiden-

¹Gmail does not include external libraries and does not mix content from HTTP sites; nevertheless, several other web applications we study in Section 6.3 include external libraries and mixed content, which poses serious threat to sensitive information.

tiality of the sensitive corporate emails must be maintained by authenticated encryption. Other examples include credit card numbers, contents of shopping carts in e-commerce applications, user passwords in login pages, payee accounts in online banking sites, and so on. Our assumption is that preventing malicious code from running in the complex web browser is not tractable; however, we aim to prevent unauthorized leakage or tampering of sensitive data.

Controlled operations. If we isolate sensitive data from all computations, it would severely disincentivize web applications from outsourcing rich functionality to the client-side browser. There is a trade-off between the information isolation guarantees and the richness of functionality that can operate on sensitive information. In our Gmail example, Alice should be able to use Gmail’s interfaces on her corporate emails for reading, composing, formatting (configuring fonts, colors, HTML formatting or alignments), archiving, forwarding and so on. For practical usage, we aim to allow such owner-specified operations to compute output from sensitive data. However, such output must be protected from confidentiality and integrity attacks from all other code in the browser, and thus is encrypted by default. The data owner (e.g., Gmail) can also specify which operations reveal computed information; however, we leave the design and choice of these functions up to the owner. In this work, we make encryption possible on simple data types, such as strings and integers, to limit the TCB size — of course, more complex data (such as arrays, formatted HTML) can still include encrypted data in them (see Section 6.2.4).

Certified user inputs. If the browser and web application code are not trusted, user inputs, such as keyboard inputs and mouse clicks, can be easily changed before they are processed by code in the TCB. In this work, we focus on enabling users to enter sensitive text strings via a trusted keyboard. Specifically, we provide a trusted path for user inputs between the OS keyboard events and the trusted web application event handlers (specified as CRYPTON functions), through the untrusted web browser. Our focus on keyboard events is guided by the observation that a significant fraction of sensitive data on the web pertains to user keyboard inputs. In our Gmail example, Alice signs in by using her password, composes a sensitive email by typing, and searches for keywords also by using the keyboard.

Proof of impression. Malicious client-side code can attempt to disable critical messages that are supposed to warn users of potential threats. For example, the webmail server may show a warning for emails suspected to be scams and banks may want to warn of fraudulent activities. As another example, when the Gmail servers detect suspicious login activities, such as simultaneous logins from different geolocations, it alerts users to change their passwords. Such messages are crucial, and may be helpful for users to prevent attacks or to stop them at an early stage.

In this work, we focus on providing a “proof of impression” of simple data types such as strings and byte streams. This includes verifiable rendering of string content and

bitmaps, during which rendering of other untrusted content into the same region is temporarily blocked for t seconds ($t=3$ by default) [73]. Such a guarantee is useful beyond the Gmail example, such as online advertisements (ads). Proof of impression can help advertisers that bill publishers for ad impressions to disambiguate real ad impressions from ad fraud with “laundered” traffic [47].

6.1.3 Design Overview

Our design changes the monolithic trust model where web servers trust the entire browser (including add-ons) and client-side application code. In our design, developers encapsulate sensitive data, such as corporate emails or bank statements, into CRYPTONS and directly embed them in HTML documents. To enforce the security guarantees and semantics of CRYPTONS, the web server trusts a small, standalone engine at the client called the CRYPTON-KERNEL. The CRYPTON-KERNEL executes outside the web browser and is protected from a compromised browser by standard OS sandboxing mechanisms, similar to those used by existing browsers such as Google Chrome [130, 129]. A CRYPTON-compliant web browser recognizes CRYPTONS during HTML document parsing and delegates their handling to the CRYPTON-KERNEL, as detailed in Section 6.2.

The CRYPTON-KERNEL acts as a root-of-trust to protect sensitive CRYPTONS at the client-side. Specifically, CRYPTON-KERNEL builds three trusted paths passing through the untrusted browser: (a) a secure communication channel between the web server and itself, (b) a trusted path from the CRYPTONS to the user’s screen (i.e., to the software rendering pixel maps or the GPU display buffers), and (c) a trusted path between the user’s keyboard inputs to CRYPTONS functions that handle keyboard events. These trusted (data) paths run through the untrusted browser code, reusing existing browser functionality. We design our system to use memory encryption to protect data on these trusted paths using authenticated encryption at the granularity of memory bytes [95].

Secure Channel To Server. At the start of a web session exchanging sensitive data, the web server establishes a secure channel with the trusted CRYPTON-KERNEL. This secure channel enables the web server to share a session key set \mathcal{K} with the CRYPTON-KERNEL, used by both parties to encrypt/decrypt CRYPTONS in the session. To avoid bloating our TCB, the CRYPTON-KERNEL delegates all network communication to the untrusted browser. Thus, it is important to establish a secure channel against man-in-the-middle attacks, in case the browser gets compromised. We use the standard mutual SSL authentication protocol to establish this channel [54]². SSL certificate verification for server authentication is a common, backwards compatible mechanisms for HTTPS sites today. To enable the server to authenticate the CRYPTON-KERNEL’s certificate, we assume the CRYPTON-KERNEL uses a certificate self-signed by an RSA private

²An alternative is to use a shared symmetric key between the server and the CRYPTON-KERNEL. However, we choose the SSL-based authentication for easier implementation, as we can reuse off-the-shelf libraries.

stored in the CRYPTON-KERNEL. The corresponding public key is uploaded by the user to the server via an out-of-band secure channel (or via “first-time trust” mode as in SSH [170]). Relying on users to upload public keys is a common practice used by web services such as Github [59] and BitBucket [21], so we expect a similar user experience for using a CRYPTON-enabled website.

Secure Display. Sensitive content, such as corporate emails, must remain encrypted in the untrusted browser until the final rendering of the content. The CRYPTON-KERNEL provides a trusted data path between the CRYPTON content received from the server to the GPU buffer [15], which finally render bitmaps and pixels to the screen. This trusted path is implemented as part of the CRYPTON-KERNEL. To enable the browser’s functionality, sensitive content must be given opaque access to the untrusted browser, say for deciding the layout dimensions or for storing them in the DOM or untrusted JavaScript heaps. To enable this functionality to work, the CRYPTON-KERNEL encapsulates encrypted sensitive content into *opaque objects* before passing them to the web browser. These opaque objects (e.g. encrypted string class) allow the browser to process sensitive data as opaque blobs of text, determine the length of the underlying plaintext for layout, but do not permit any malicious operation that would leak plaintext data. Opaque objects are finally rendered by the CRYPTON-KERNEL using a “sandboxed GPU buffer” mechanism [15], which enables it to ensure that content is rendered on top for at least t seconds.

Trusted Path For User Input. For sensitive content generated at the client, such as by user keyboard inputs, the CRYPTON-KERNEL provides a trusted path between the OS keyboard events to the CRYPTON functions designated to handle input events. To enable these, CRYPTON-KERNEL intercepts all keyboard events from the OS, similar to Chrome’s sandboxing mechanism [15]. It uses opaque string objects to establish the trusted data paths via the untrusted browser code.

We expect the user Alice to recognize and enter sensitive keyboard inputs (e.g., passwords) through the CRYPTON-enabled trusted path. To help users securely interact on and across multiple CRYPTON-enabled sessions, CRYPTON-KERNEL displays 2 security indicators above the untrusted browser UI. For distinguishing when her user inputs are going to be encrypted (vs. when she is interacting with a non-CRYPTON-enabled session), CRYPTON-KERNEL displays a user-selected secret image icon. This image icon states that the user’s key events will be encrypted until Alice invokes a special pre-selected key sequence to signal end of the encrypted keyboard session. This indicator informs users whether their sensitive keystrokes are going to be encrypted; we expect users to only enter sensitive information when seeing the right image icon.

Second, CRYPTON-KERNEL displays the URL of the website whose keys are being used to encrypt the keystrokes, in a special unspoofable URL bar above the browser chrome. This allows Alice to distinguish to which of the multiple CRYPTON sessions her present keystrokes are being delivered. This indicator informs the user of the in-

tended recipient of the encrypted keystrokes; users are expected to verify this URL indicator and ensure that it is the right recipient for the sensitive data they are about to enter.

Both UI indicators are securely rendered to the screen directly by the CRYPTON-KERNEL and can not be overlaid by the attacker. The CRYPTON-KERNEL does not allow the untrusted browser to inject key sequences or read sensitive GPU buffers using standard sandboxing techniques. We discuss more details in Section 6.2.

Assumptions. A CRYPTON-compliant server uses an HTTPS frontend as the server-side interface to the CRYPTON-KERNEL. HTTPS frontends often handle authentication and are common for load balancing, before delegating the session to backend server — we believe our design can be deployed in existing HTTPS frontends.

CRYPTON-compliant servers must be careful to default-deny access to sensitive content outside the secure channel established with a trusted CRYPTON-KERNEL. Otherwise, this would incentivize attackers to fool users into accessing the sensitive content over non-CRYPTON-enabled sessions.

To protect the CRYPTON-KERNEL, our solution leverages OS processes and system call sandboxing to isolate the untrusted browser from the CRYPTON-KERNEL. Robust sandboxing mechanisms are known and used by existing browsers such as Chrome [15]. For additional protection, especially in emerging Web OS stacks [112, 127], hardware-assisted dynamic root of trust measurement (DRTM) (such as those provided by Intel TXT [78, 117]) can be used to ensure the dynamic code integrity of the CRYPTON-KERNEL.

6.1.4 Out-of-scope Threats

Our focus is on providing integrity and confidentiality of data. So, for example, our defenses cannot be used to protect data used in server-side authorization such as CSRF tokens, session cookies and authentication tokens (for single-sign on mechanisms), as blocking these attacks require ensuring the authenticity of the sender of sensitive data. Similarly, browser vulnerabilities may be used to completely deny access to sensitive data (denial-of-service); our defenses do not enforce data availability to the intended recipient.

In our design, a trusted component (the CRYPTON-KERNEL) acts a root-of-trust on the client device. The CRYPTON-KERNEL establishes a trusted path to receive sensitive user input (via keyboard inputs). There are several attacks that can elicit sensitive information from the user *outside* the trusted paths, say by utilizing phishing [181, 186, 58], confusing the user with fake security indicators [178, 80], exploiting time-of-check-to-time-of-use (TOCTTOU) windows in our security indicators, clickjacking [11, 73], shoulder surfing or through social coercion [23]. We recognize that these are important channels of information loss to consider for guaranteeing end-to-end security; however, defenses for these specific attacks are of independent interest. We admit that a usability evaluation of our additional indicators would be useful before a full deployment;

however, such an evaluation would merit from a separate research.

Finally, we point out that our design requires developers to carefully design CRYPTON functions to avoid side-channels via timing and control flow, which are declassification interfaces to sensitive data. These are well-studied classes of attacks on information flow and encryption systems [33, 24, 30]; tools to detect and minimize the capacity of side channels are useful for developers [90].

6.2 Design

We propose a novel data protection solution with CRYPTONS that can be directly programmed into web pages.

6.2.1 CRYPTON Definition

The CRYPTON abstraction incorporates sensitive data protected with integrity and confidentiality, the operations permitted to decrypt and process the information, as well as the functional *policy* that determines which keys to use for decrypting sensitive information and for encrypting outputs.

More formally, we define a CRYPTON as a 5-tuple $\mathcal{W} = (\vec{\mathcal{D}}, \vec{\mathcal{F}}, \mathcal{U}, \mathcal{V}, \mathcal{I})$. As defined later, $\vec{\mathcal{D}}$ is a sequence of information blocks and $\vec{\mathcal{F}}$ is a sequence of CRYPTON functions. \mathcal{U} is the web address URI of the CRYPTON-enabled web server, the HTTPS frontend of the web server. The external verifier \mathcal{V} is the HTTPS web URI accepting proof-of-impression tokens. \mathcal{I} is a unique identifier (ID) for the CRYPTON.

A CRYPTON \mathcal{W} syntactically binds permitted operations and sensitive data in its definition; this binding is signed by the SSL public key corresponding to \mathcal{U} and is verified and maintained by CRYPTON-KERNEL. When the CRYPTON-KERNEL first processes a CRYPTON, it communicates with \mathcal{U} over a secure channel and fetches a set of symmetric encryption keys \mathcal{K} corresponding to that specific CRYPTON. Note that CRYPTON keys \mathcal{K} are not sent as part of \mathcal{W} , which would otherwise permit them to be read by adversaries. The set \mathcal{K} consists of a default encryption/decryption key κ_0 , an HMAC key κ_{hmac} , and optionally other keys $\kappa_1, \dots, \kappa_n$, for authenticated encryption using symmetric key ciphers (256-bit AES-GCM). These keys are kept secret and known only to the CRYPTON-KERNEL; they are referenced by their key indices in CRYPTON functions and information blocks as explained below.

Conceptually, each CRYPTON function \mathcal{F}_i is a 3-tuple $\mathcal{F}_i = (\mathcal{P}, \mathcal{R}, \tau_{\mathcal{F}})$. The syntactic definition of \mathcal{F}_i binds tuple elements together; these bindings are signed to ensure their integrity. $\mathcal{P} : \mathcal{K} \times \mathcal{K}$ is a set of policies defined for each \mathcal{F}_i over keys in $\mathcal{K} \cup \{\perp, \kappa_{int}\}$, where \perp means public (no encryption), and κ_{int} is a CRYPTON-KERNEL-specific key to protect user inputs, such as keystrokes. Such a policy dictates how a CRYPTON function encrypts and decrypts sensitive data. A policy $\kappa_1 \rightarrow \kappa_2$ for a function \mathcal{F}_i indicates that \mathcal{F}_i decrypts sensitive data using the key κ_1 . Global or local variables written during the function execution, arguments of calls to the untrusted browser, as well as return values, are encrypted with the key κ_2 . Policies are optional; the default policy enforced

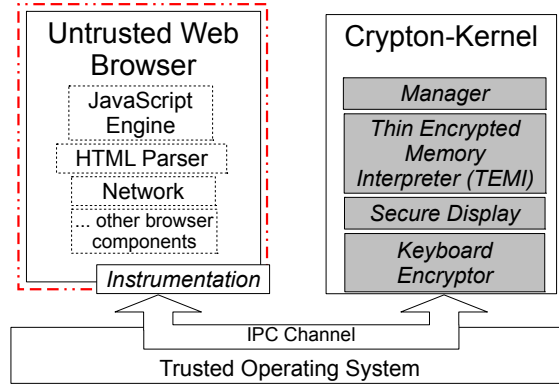


Figure 6.1: Overview of the CRYPTON-KERNEL, consisting of 4 components in gray, and the sandbox of a dashed red rectangle.

is $\kappa_0 \rightarrow \kappa_0$, i.e., decryption and encryption both with the default key. We explain the semantics of the execution of each function in detail later. All values computed in implicit or explicit flows from CRYPTON functions are encrypted with CRYPTON keys. \mathcal{R} optionally lists the arguments to the function and the return value. Finally, functions have a type field $\tau_{\mathcal{F}}$ which separates 2 kinds of functions: (a) a proof-of-impression (POI) function that must be invoked to handle impression tokens, and (b) all other functions. POI functions compute encrypted messages over the nonce token and the impressions of sensitive messages and send them to the external verifier \mathcal{V} .

An information block is a 6-tuple $\mathcal{D}_i = (\mathcal{I}, \kappa, \mathcal{I}_{\mathcal{D}}, \mathcal{A}, \tau_{\mathcal{D}}, \mathcal{F}_{\mathcal{D}})$. \mathcal{I} and $\mathcal{I}_{\mathcal{D}}$ are identifiers for the owner CRYPTON and the information block, respectively. The encryption key for \mathcal{D}_i is specified by $\kappa \in \mathcal{K}$. \mathcal{A} is the encrypted ciphertext of the enclosed sensitive data. Finally, if a certain information block requires proof-of-impression for its render value \mathcal{A} , the field $\tau_{\mathcal{D}}$ indicates the nonce token for this information block, and $\mathcal{F}_{\mathcal{D}}$ refers to the function to be invoked with the impression token by CRYPTON-KERNEL.

6.2.2 CRYPTON-KERNEL Design

The CRYPTON-KERNEL provides secure processing of sensitive web content for a CRYPTON-KERNEL-compliant web browser, while never exposing decrypted data to the browser. Our solution needs to instrument the untrusted browser, which communicates with the CRYPTON-KERNEL via IPC channels. We present a brief overview of our solution with a hypothetical example demonstrating protecting sensitive emails in a webmail application (Figure 6.2).

When the CRYPTON-KERNEL-compliant web browser parses an HTML document, such as the sample web page in Figure 6.2, it encounters the CRYPTON header, functions or information blocks. At this time, the browser invokes the CRYPTON-KERNEL via an IPC interface. Once invoked, the Manager component of the CRYPTON-KERNEL processes these special tags, including validating their integrity, and retrieving CRYPTON keys via a secure channel from the server URLs embedded in the CRYPTON header. The Manager component returns opaque objects to the untrusted browser for processed

```

<html>
  <head>
    <cryptonheader
      owner='https://www.gmail.com'
      https_frontend_url=
        'https://www.gmail.com/crypton_keys.json'
      verifier_url=
        'https://www.gmail.com/crypton_verifier'
      iv='fVrXgz3pQ2kL7gHW...'
      hmac='DtTjG0vEvJ4YkVqF...' ></cryptonheader>
    ...
    <script>
      [crypton] function searchEmail(keywords) {
        'fun_body':
          '{ var words = keywords.split(...); ...}',
          'policy': '$\kappa_{int} \rightarrow \kappa_0$',
          'hmac' : 'Vukhwo5WuglDmnh...'
        }
      [crypton] function parseJson(json)
        ... details omitted for brevity...
    </script>
  </head>
  <body>
    ...
    <crypton
      id='ib_txt'
      owner='https://www.gmail.com'
      key='$\kappa_0$'
      sensitive-data='Wil jGq2M69E8w6xn...'
      hmac='jOiLlDboZ0G7jNAX...' ></crypton>
    <button type='button' id='email_search_btn' onclick=''searchEmail(...)' ' >
      Search</button>
    ...
  </body>
</html>

```

Figure 6.2: CRYPTON Tags Embedded in HTML. An example of search emails matching user-input keywords, including the CRYPTON header, functions, and information blocks.

CRYPTONS. These opaque objects encapsulate encrypted values and metadata such as the corresponding CRYPTON ID and decryption key. They are propagated internally by the untrusted browser, akin to ordinary browser objects. (Step A in Figure 6.3).

The untrusted browser handles the layout of web content, and composes intermediate constructs for rendering. Another CRYPTON-KERNEL component, called the Secure Display, intercepts rendering requests from the untrusted browser, and converts the layout constructs into pixel maps written into the GPU buffer. When opaque objects enclosing encrypted texts traverse through untrusted code paths in the browser, and finally reach the Secure Display component, the Secure Display decrypt them before transforming them into pixel data for the GPU buffer. For example, sensitive emails in our example are encrypted when they are downloaded into the browser. When they are being rendered, the Secure Display component of the CRYPTON-KERNEL decrypts them into plaintext that is displayed to users (Step B).

To protect user inputs, the Keyboard Encryptor component of the CRYPTON-KERNEL encrypts all user inputs for the CRYPTON-enabled webmail application. For example, it encrypts the search keywords while the user enters them in the untrusted browser. The resulted opaque object containing the encrypted user inputs is finally dispatched into the

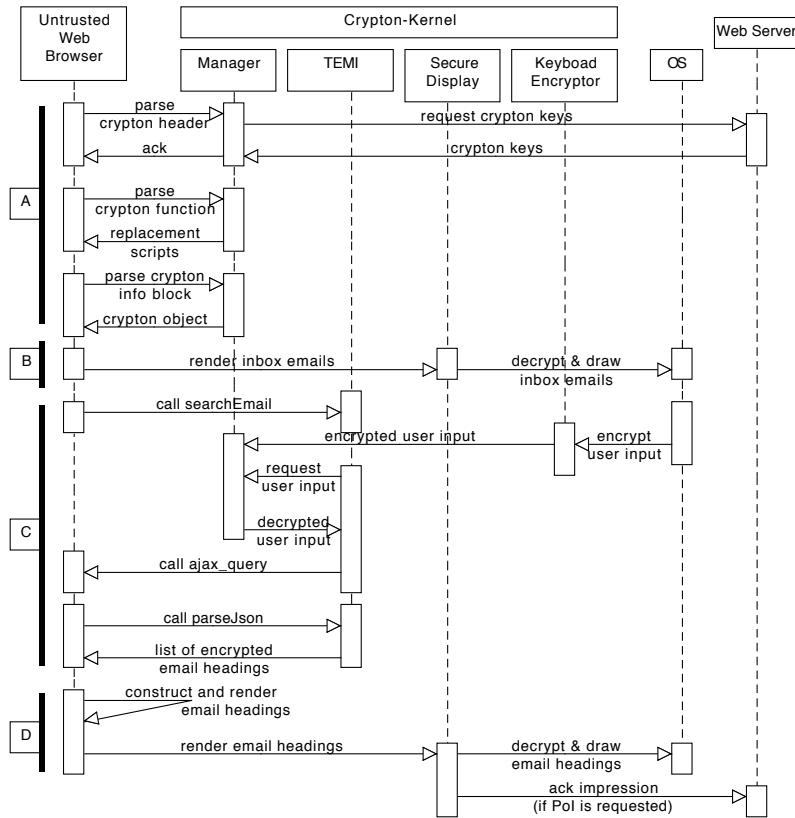


Figure 6.3: Sequence Diagram for CRYPTON-KERNEL Operations

event handler function `searchEmail` triggered by a user click on the “Search” button. The webmail developer marks `searchEmail` as a CRYPTON function with a policy $\kappa_{int} \rightarrow \kappa_0$, allowing it to decrypt user inputs (such as search keywords) encrypted with the CRYPTON-KERNEL’s internal key κ_{int} . When `searchEmail` is invoked, the untrusted browser transmits the execution to a reduced execution environment in the CRYPTON-KERNEL, called the Thin Encrypted Memory Interpreter (TEMI). The TEMI decrypts the search keywords encrypted in the opaque event object during the execution of `searchEmail`. This allows the function to preprocess the search terms, such as replacing ‘+’ with ‘ADD’, and encode the texts. Then `searchEmail` constructs an XMLHttpRequest containing the preprocessed search terms and sends it to the webmail server. When the webmail server receives the request, it decrypts the search terms and search for them in the email database. (Step C).

Subsequently, search results are returned to the untrusted browser in JSON format where innermost email headings are encrypted. The registered XMLHttpRequest handler function `parseJson` is invoked to tokenize and validate the JSON string, and return a composed HTML fragment to be rendered to the user. The email headings in the HTML fragment remain as opaque objects to untrusted code in the browser, and the Secure Display decrypts them when rendering the headings for the webmail user (Step D).

6.2.3 Security Invariants

To protect the confidentiality and integrity of sensitive web content in an untrusted browser, our solution enforces the following semantics as security invariants for CRYPTONS.

For information isolation:

P0: Maintaining Syntactic Bindings. A CRYPTON syntactically binds its elements to each other (such as \mathcal{K} to \mathcal{I} , \mathcal{P} to each \mathcal{F}_i and so on) with mentioned keys. These bindings are checked and enforced throughout the lifetime of the CRYPTON. Authenticated encryption ensures that any attempt to tamper with such bindings by malicious code will be detected.

P1: Secure Storage. All CRYPTON information blocks, functions and intermediate computation by CRYPTON functions is stored in private memory outside the browser, or encrypted when stored in memory shared with the browser. The standard guarantees of authenticated encryption apply to these.

P2: Secrecy of \mathcal{K} . The keys \mathcal{K} are only stored in the CRYPTON-KERNEL and not accessible to the CRYPTON-compliant browser.

For controlled operations:

P3: Functional Policy. If a function \mathcal{F}_i is bound to a policy $\kappa \rightarrow \kappa'$, then all sensitive data processed by it are decrypted using κ , and all values written by \mathcal{F}_i are encrypted with κ' , e.g., modified global variables, local variables, arguments of calls to untrusted browser functions, and return values. See **P5** as the only exception.

P4: Non-interfering Execution. The execution of two functions \mathcal{F}_1 and \mathcal{F}_2 are non-interfering on the CRYPTON-KERNEL, if they are bound to policies $\kappa_1 \rightarrow \kappa'_1$ and $\kappa_2 \rightarrow \kappa'_2$ respectively, and $\{\kappa_1, \kappa'_1\} \cap \{\kappa_2, \kappa'_2\} = \emptyset$.

P5: Information Release. Only functions bound to policies $_ \rightarrow \perp$ output plaintext to the browser³.

For certified user inputs:

P6: User Input Encryption. User keyboard inputs are encrypted with the CRYPTON-KERNEL's internal key κ_{int} , and only CRYPTON functions bound to policy $\kappa_{int} \rightarrow _$ can decrypt the inputs.

³ $_$ denotes any key or \perp

P7: Restricted Keyboard Access. The CRYPTON-KERNEL intercepts all keyboard events before they are delivered to the browser.

For proof of impression:

P8: Proof of Impression. When an information block \mathcal{D}_i is *rendered*, it should be displayed for a certain period of time without obstruction or overlay by other UI content. The POI function specified by the developer is invoked to compute an encrypted message acknowledging the rendering to the verifier \mathcal{V} in a secure channel.

P9: Restricted Display Access. The CRYPTON-KERNEL mediates the rendering operations initiated by the browser.

With these security invariants, CRYPTONS ensure the confidentiality and integrity of enclosed sensitive data throughout their lifecycle in untrusted web browsers.

6.2.4 Key Techniques and Security Analysis

To ensure the security invariants above, the CRYPTON-KERNEL provides a reduce engine for permitted operations on sensitive data, and interposes browser display and keyboard events to support trusted paths for display and user inputs.

Thin Encrypted Memory Interpreter. To execute controlled operations on sensitive data, we design the Thin Encrypted Memory Interpreter (TEMI) — a reduced scripting engine. In designing TEMI, we face a tradeoff between supporting rich functionality and minimizing the TCB. A full-fledged JavaScript engine typically has 200,000 (JavaScriptCore) to 400,000 (the V8 engine) LOC. However, we observe most of potential sensitive data types are integers and strings, which accounts for only a very small fraction of the entire JavaScript engine. Of course, these primitive types are aggregated into higher-level abstract data types (such as arrays, lists, objects, and so on), and much of the logic that operates on the higher-level data types is agnostic or independent of the inner nested data. By leveraging this observation, we design a “thin” TEMI, which has a TCB of only 13,000 lines of code.

The TEMI runs inside the CRYPTON-KERNEL’s process, communicating with the process(es) of the untrusted web browser via IPC messages (shown in Figure 6.4). The TEMI has a private virtual register set (for local computation) in its private memory region. It decrypts sensitive data in opaque objects when they are loaded into virtual registers, and re-encrypt all values written from virtual registers to opaque objects, including global variables, arguments, and return values. By default, the TEMI uses the default CRYPTON key κ_0 for encryption and decryption; a developer-specified policy $\kappa_1 \rightarrow \kappa_2$ of a CRYPTON function dictates κ_1 used for decryption, and κ_2 for encryption (**P3**). A CRYPTON function that is allowed to disclose information has a policy $\perp \rightarrow \perp$ (**P5**). The TEMI retrieves CRYPTON keys from CRYPTON-KERNEL’s private memory storing keys transferred from the web server (**P2**).

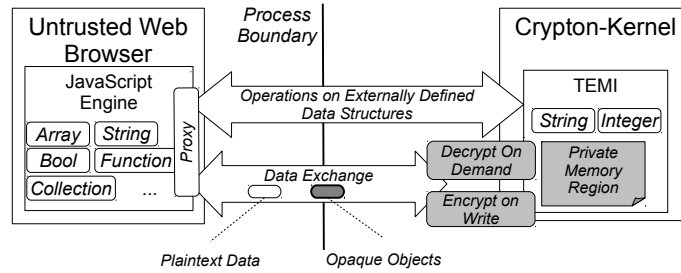


Figure 6.4: TEMI: a Reduced Execution Environment for CRYPTON Functions to Access Sensitive Info

During the execution of a CRYPTON-KERNEL function, any access to higher-level abstract data types results in an IPC message to a proxy in the JavaScript engine of the untrusted browser with opaque objects as arguments. The proxy translates the message into operations that are executed in the untrusted JavaScript engine. For operations initiated by the CRYPTON-KERNEL, any string and integer operation that emerges during execution in the untrusted JavaScript engine is tunneled back to the CRYPTON-KERNEL for processing opaque objects. In our running example in Figure 6.2, the `parseJson` function sends a message to the untrusted browser to access complex data structures, such as arrays to perform parsing operations. The execution then switches back to the TEMI when an `Array.toString` function is invoked so that operations on sensitive strings can still be processed in the decrypted form. This unmodified CRYPTON function runs in our TEMI as it switches back and forth between the untrusted browser and the CRYPTON-KERNEL. For data exchange, the TEMI and the untrusted browser passes integer and string data in the IPC messages, while only pass references to high-level data structures as they are processed exclusively at the untrusted browser side.

In our design, accessing higher-level data types defined in the untrusted browser from the TEMI incurs no explicit data leakage, since all arguments are encrypted (**P1**). Implicit control flow leakage is possible, while we assume it can be largely ignored in legitimate code [30], such as CRYPTON functions. For compatibility with existing browsers, when one process is running JavaScript, we halt the other process until the running process returns.

To prevent interference between different CRYPTON functions, the TEMI executes each CRYPTON function in a separate context, which is destroyed upon returning of the function (**P4**). This includes virtual registers stored in the private memory region, as well as encrypted intermediate variables written in the shared memory region. Similar to garbage collectors, the TEMI clears all these values when a CRYPTON function returns.

Secure Display. The CRYPTON-KERNEL allows automatic decryption of opaque objects when they are being rendered, but prevents untrusted browser code to access the decrypted information. The Secure Display component of the CRYPTON-KERNEL can use sandboxing techniques similar to Google Chrome [15], such as restricted tokens and Windows job and desktop objects on Windows [130] and `seccomp` on Linux [129], to intercept untrusted browser’s access to the GPU buffer and graphic libraries on the OS

(P9). To render any UI content, the untrusted browser must pass the rendering requests to the Secure Display. The Secure Display in turn converts web contents and browser widgets into intermediate rendering constructs (such as `Glyphs` that are basic shapes in text rendering [27]) and sends them to the graphics library for rendering. During this process, the Secure Display decrypts opaque objects containing sensitive strings, with the keys specified in the opaque objects. To prevent UI obstruction, while rendering such opaque objects, for t seconds, the Secure Display temporarily suspends other rendering requests to the same destination GPU buffer location. It also invokes the `PoI` function for the opaque string being rendered, if applicable. The `PoI` function then computes an acknowledgement token encrypted with the CRYPTON’s default key κ_0 and sends it to the external verifier (P8).

Keyboard Encryptor. The Keyboard Encryptor interposes on keyboard events from the graphics toolkit that receives keystrokes. For each keystroke event, it encrypts it with a CRYPTON-KERNEL-specific internal key κ_{int} before it reaches the web browser code (P6, P7). Therefore, web applications cannot read the clear-text of user inputs unless by calling the CRYPTON functions with a policy like $\kappa_{int} \rightarrow \dots$. Together with P3, this ensures that keyboard inputs are protected, and are available only to intended functions.

Resilience against malicious code in web browsers. Although the CRYPTON-KERNEL closely integrates the untrusted web browser, the security guarantee ensured by the CRYPTON-KERNEL are independent from the untrusted web browser, i.e., all disclosure of the sensitive information is controlled by CRYPTON functional policy.

To allow non-invasive operations that do not require plaintext, the encrypted data in CRYPTON information blocks are stored on the browser’s storage as opaque objects. These objects are protected by authenticated encryption for their integrity and confidentiality. The keys are available only to the CRYPTON-KERNEL. Untrusted code has no access to it (P2), and thus cannot decrypt sensitive data, unless by explicitly invoking information releasing CRYPTON functions (P5). User inputs are automatically encrypted before they reach the untrusted web browser (P6), so they are under the same protection as sensitive data coming from web servers. Web servers can also verify sensitive messages are properly displayed via proof of impression (P8).

Although malicious code cannot decrypt the sensitive data in ciphertext, it can tamper with them. However, this will be detected by integrity verification built into authenticated encryption. This ensures the integrity of all CRYPTONS from the web and sensitive user inputs at the client.

6.2.5 Prototype & Deployment

We implement a prototype of the CRYPTON-KERNEL integrated with WebKit-GTK (rev 45311) with JIT disabled, for the ease of implementation. Figure 6.1 closely resembles the design of our prototype. Our prototype reuses the OpenSSL code for AES-GCM

App Name	Sensitive Data	Server-side Change	Client-side Change	Estimated Total Conversion Effort
RoundCube	Mails with subject marked as “[sensitive]”	8 LOC php	9 Functions, 0.5K LOC JS	2-3 Man-Days
AjaxIM	Instant messages	6 LOC php	9 Functions, 0.4K LOC JS	2-3 Man-Days
WordPress	Blogs containing “[sensitive]” in titles and search keywords	10 LOC php	19 Functions, 1.2K LOC JS	4-5 Man-Days

Table 6.1: Summary of Case Studies on 3 Popular Web Apps, demonstrating modest adoption effort

encryption/decryption. We intercept signal dispatch from the GTK+/Glib to the browser to encrypt user inputs. As user input events are fired for each keystroke, we use a stream cipher to encrypt user inputs. Thus, positions of user input characters are maintained as original, and this naturally supports mouse text selection. When user inputs are sent to web servers, we use a CRYPTON function to re-encrypt them with the AES-GCM blocker cipher with the specified key. Table 6.2 lists the estimated sizes of components in our design of the CRYPTON-KERNEL. Comparing to a web browser, such as Firefox and Chromium, which typically has around 800K to 1,100K lines of source code, the TCB in our unoptimized prototype is about 40-50 times smaller.

CRYPTON-KERNEL Component	LOC
Manager	2.9K
TEMI	12.7K
Secure Display	4.8K
Keyboard Encryptor	0.5K
Total	20.9K

Table 6.2: Estimated Size of TCB in CRYPTON-KERNEL Prototype

Our prototype implements functionalities sufficient to support our studies with real-world web applications. In a full deployment, several browser components need to be modified to propagate the opaque objects with encrypted data, including support for untrusted browser sandboxing, browser spell checkers, all CSS styling features, web page printing, WebGL and GPU accelerations. Such support has not been implemented in our current prototype. For deployment into real web browsers, we consider an alternative deployment into Google Chrome’s browser kernel, leveraging existing privilege separation and sandboxing mechanisms in Google Chrome. This alternative deployment would move the code of browser renderers out of the TCB for data protection.

6.2.6 Alternative Deployment into Google Chrome

Our current design only requires a very small TCB completely outside web browsers. However, it requires extensive changes to existing web browsers. We also consider an alternative deployment scenario with slightly larger TCB but smaller adoption cost into Google Chrome. Google Chrome partitions more vulnerable components into the renderer processes, and leave more security-sensitive components in the browser kernel process. It has also implemented sandboxing mechanisms to prevent renderer processes from directly accessing UI rendering, user inputs, file systems, and network. All such access has to go via interfaces exposed and checked by the browser kernel. Thus,

it is straightforward to implement the CRYPTON-KERNEL into the browser kernel of Chrome, leveraging the existing UI and user input sandboxing mechanisms. Under this alternative deployment, the reduction of the code size that can access sensitive data is still significant. It removes more than 900K lines of code in renderer processes and all client-side code in web applications out of the TCB for data protection.

6.3 Evaluation

We apply our solution to real-world web applications to study the applicability and adoption cost. We first manually convert 3 popular web applications to use our solution to protect typical sensitive data in those applications, and then extend our study to Alex Top 50 web pages. Both micro and macro studies show the effectiveness of our solution in protecting typical sensitive data on the present web with modest adoption effort required.

6.3.1 Applicability to Real-world Applications

Micro-study on open-source web applications. We perform case studies on 3 open-source web applications to measure 2 aspects, i.e., *a*) how effectively our solution can protect sensitive content in real-world applications, and *b*) how much developer effort is required for adopting our solution. We choose applications of different categories: RoundCube [157], a webmail server, AjaxIM⁴, a web-based instant messenger, and WordPress [56], a web blog service. We manually convert the source code of the 3 applications by: 1) modifying the server-side code to encrypt sensitive content before sending it to clients; 2) identifying client-side JavaScript functions that need to decrypt sensitive data for operations and converting them into CRYPTON functions; and 3) rewriting client-side JavaScript functions that receive and process user inputs into CRYPTON functions; these CRYPTON functions encrypt user inputs before sending them to web servers. We check all tests with a server-side proxy.

We find that with modest effort in converting these applications, our solution can effectively protect typical sensitive data, such as sensitive emails, instant messages, blog entries and comments. We write a 450-line custom PHP library for common functionalities to process CRYPTONS in PHP applications. We manually rewrite the 3 web applications, leveraging the custom library to wrap sensitive web content into CRYPTONS. Table 6.1 summarizes the results of our case studies on the three applications. Typical client-side operations on sensitive data we observe include trimming whitespaces in strings, serializing HTML content, emotion text replacement, URI encoding, etc. We mark them as CRYPTON functions in our experiments to support the legitimate functionalities. For brevity, we leave out detailed steps here, and a summary of our modification to application source-code is available online [8].

⁴We use the AjaxIMRPG fork [40] that is better maintained than the trunk.

Web App Detail	Sensitive Info	# & Size of Functions Requiring Decrypted Sensitive Info [Percentage of Total Code Size]	# & Size of All JS Functions	# Browser-TEMI Interactions vs. # All Access to JS Data Types
Gmail Login	Username, Password	2 (0.29 KB) [0.43%]	196 (66.5KB)	0.7% (956/141492)
Gmail Compose Email	To, Subject, Content	9 (1.84KB) [0.98%]	745 (186.8KB)	0.01% (89/541883)
Gmail Read Email	From, Subject, Content	2 (1.1KB) [0.64%]	730 (171.1KB)	0.02% (92/390206)
Ask Search	Search Terms	3 (1.3KB) [0.31%]	569 (416.2KB)	0.01% (40/218922)
Google Search	Search Terms	6 (0.92KB) [0.26%]	581 (352.9KB)	0.2% (437/206675)
Google+ Post	Post Content	5 (1.01KB) [0.13%]	1150 (750KB)	0.005% (48/947254)
Netflix	Username, Password	1 (0.6KB) [0.2%]	419 (289.9KB)	0.1% (170/162853)
IMDb Search	Search Terms	1 (0.24KB) [0.02%]	1131 (1.1MB)	0.1% (422/323208)
Facebook Read Post	Post Content	1 (0.5KB) [0.18%]	730 (268KB)	0.02% (347/1216952)
Facebook Friend Search	Search Terms	9 (1.7KB) [0.50%]	959 (336.6KB)	6.4% (60305/937108)
eBay Item List	Item's Description	8 (1.49KB) [0.62%]	725(239.2KB)	0.1% (326/241901)
Amazon Login	Username, Password	2 (1.87KB) [1.28%]	240 (146KB)	0.5% (1629/321275)
Amazon Search	Search Terms	9 (1.77KB) [0.24%]	485 (732.7KB)	0.1% (544/433110)
MSN Search	Search Terms	5 (1.4KB) [0.08%]	1185 (1.7MB)	0.5% (2096/435124)
StackOverflow	Post Content	5 (2.36KB) [1.18%]	429 (198.9KB)	0.2% (830/406328)
Twitter Login	Username, Password	3 (0.61KB) [0.27%]	472 (223.6KB)	0.004% (27/622874)
Wikipedia Search	Search Terms	6 (1.9KB) [0.23%]	438 (794.1KB)	0.45% (382/84818)
Babylon Purchase	Email Address, Credit Card Info	3 (1.17KB) [0.70%]	485 (165KB)	0.6% (3232/515586)
Bank (anonymized) Login	Username, Password, PassKey	3 (1.49KB) [1.01%]	250 (146.3KB)	~0% (1/100037)
BankOfAmerica Login	Username, Password	2 (1.01 KB) [0.23%]	780 (430.4KB)	~0% (3/441534)

Table 6.3: Study of 20 Popular Web Applications. $< 1\%$ of web application code needs to run in the TEMI to access sensitive info; $< 1\%$ of all calls to JS data types trigger browser-TEMI interactions.

Macro-study on real-word web applications. To further evaluate the applicability of the CRYPTON-KERNEL to other web applications, we perform a larger-scale macro study. First, we select Alexa Top 50 web pages, and identify all of those fields that require sensitive username / password inputs for signup – 18 out of the 50 applications have signup pages (Figure 6.5). We choose these applications because they often have client-side checking code on sensitive passwords (e.g. checking strength requirements). Next, we select 20 popular web applications of 5 categories, and identify scenarios where the CRYPTON-KERNEL can strengthen security against real attacks:

- *Web search pages.* We select sites that allow user to search terms. We mark search terms as sensitive because leaking search terms may permit third-party tracking.

- *Social networking sites.* These sites can be used to exchange private messages, or post comments that may be politically-sensitive. Hence, we mark posts and comments as sensitive.
- *Banking sites.* Banking sites are prime targets of browser-based attacks and several malware disguise as browser extensions. We select a local bank (anonymized for submission) that uses additional authentication mechanisms such as one-time PassKeys. In these bank web pages, username, passwords and the one-time PassKey are marked as sensitive. The one-time PassKey is interesting because banks are increasingly considering this as a second-factor authentication beyond long-lived passwords.
- *E-commerce sites.* We test eBay, Amazon and Babylon online commerce sites. On eBay, we mark one auction listing created by a seller as sensitive. Different auction listings can be loaded in the same page, and each may contain its own JavaScript. If the listing is not protected, the script from one seller may deface or tamper with that from another seller. We also consider Babylon, since users can purchase items without creating an account with Babylon. Therefore, by tampering with the purchaser’s email in the browser, an attacker can have a software license being purchased be delivered to that attacker’s email address. We mark the email address as a sensitive field on Babylon. We mark product search terms as sensitive for Amazon.
- *Gmail.* It is a rich webmail client used as our running example. We mark search terms and certain emails as sensitive.

We manually (with browser instrumentation) identify all functions that legitimately operate on sensitive information, including event handlers. Our analysis details are shown in Table 6.3. We assign the default policy $\kappa_0 \rightarrow \kappa_0$ to these functions, and all user keyboard inputs are marked as sensitive. We mark the Gmail spam warning message as requiring proof-of-impression. We also mark a post on Facebook as requiring proof-of-impression, which is necessary if a post is politically-sensitive and it’s owner want to evade censorship (via censoring browser plugins, say).

Results. We dynamically modify these web pages to encode sensitive contents into CRYPTONS using a proxy server. We find that these applications render correctly in our CRYPTON-KERNEL-compliant browser implementation, and the applications remain functional. This confirms our hypothesis that existing application can be easily upgraded to using CRYPTON-KERNEL’s functionality. Our test proxy decodes encrypted contents returned from the browser and checks them against the expected values. It also verifies that it receives proof-of-impression tokens.

Developer effort. Table 6.3 demonstrates that the developer effort to enable this functionality is modest. On average, only **1%** of the total functions in the applications need

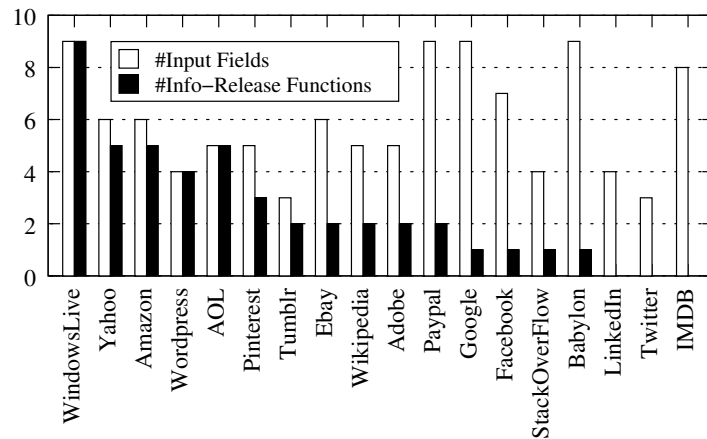


Figure 6.5: Number of User Input Fields and CRYPTON Functions Required in Signup Pages

to be included in CRYPTONS; this accounts to 2 KB of minified JavaScript per web page. The effort required to verify such code as CRYPTON functions is feasible, around **1-9** functions for each application. Figure 6.5 shows similar results that about **0-9** functions need to be verified for each signup page.

6.3.2 Reduction in Attack Surface

The isolation and controlled operations provided by CRYPTONS significantly reduce the size of code that sensitive information is exposed to. We evaluate this reduction in both web application code as well as in browser code.

Exposure to untrusted application code reduces by 99%. As we show in Table 6.3, on average less than 1% of web application code actually needs access to sensitive information, and is thus run in CRYPTON functions. All the rest of web application code only has access to opaque objects with encrypted data. On the contrary, in current web browsers, all information is exposed to the entire web application. Any malicious or compromised JavaScript library can steal the information and leak it to external parties. By isolating sensitive information into CRYPTONS, the CRYPTON-KERNEL places **99%** of web application outside the TCB.

Exposure to browser vulnerabilities reduced by 92.5%. In existing web browsers, sensitive information being processed is largely accessible to browser code running in the same process. To evaluate how the CRYPTON-KERNEL reduces such over-exposure of sensitive information to browser vulnerabilities, we study how many historical browser security vulnerabilities are prevented from affecting the security guarantees provided by the CRYPTON-KERNEL. Our study with historical security vulnerabilities in Firefox show that in total 333 of them (92.5%) cannot be exploited to violating the security guarantees provided by the CRYPTON-KERNEL. The remaining vulnerabilities either reside in our TCB (7 of them), or compromise our assumptions (another 20

vulnerabilities). The vast reduction in the exposure to browser vulnerabilities verifies the effectiveness of the CRYPTON-KERNEL in information protection. We leave more details on our methodology to Section 6.3.4 at the end of this section.

6.3.3 Performance

We run the CRYPTON-KERNEL over the 20 web applications listed in Table 6.3, with sensitive information transformed into CRYPTONS. The web sites remain responsive, without any *perceivable* slowdown. To further evaluate potential performance bottlenecks, we apply microbenchmarks to measure the performance overheads (averaged over 20 runs each) arising from encryption/decryption operations and browser-TEMI interactions when the TEMI executes CRYPTON functions.

Encryption/decryption overhead. Figure 6.6 shows the performance overhead of the CRYPTON-KERNEL compared to a vanilla WebKit-GTK browser. We use 5 CPU-intensive test pages from the Dromaeo benchmark. To vary the workload of encryption and decryption, we mark different portions of texts in the pages as sensitive, ranging from zero-sensitive pages (no sensitive text) to fully sensitive pages (all texts marked as sensitive). During these tests, the browser remains responsive and shows a maximum of 7.5% performance overhead. We also manually craft a test page to evaluate the performance of the CRYPTON-KERNEL under pathological scenarios. The performance overhead reaches 11.7% when all texts on the page are marked as sensitive, which require heavy encryption/decryption operations. Considering such performance overhead is measured from an initial unoptimized prototype, it is reasonable.

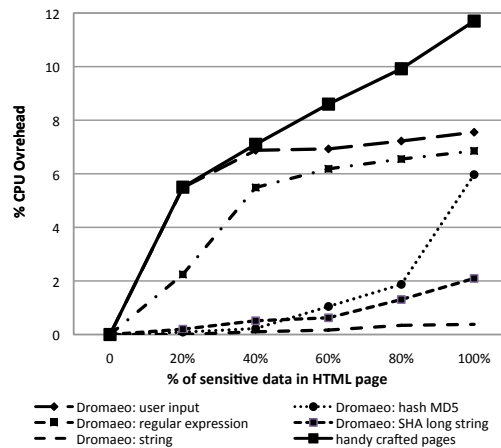


Figure 6.6: Performance Overhead of CRYPTON-KERNEL with Dromaeo Benchmarks & Manual Test Page, with varying portions of sensitive texts

Overhead of browser-TEMI interactions. We run the CRYPTON-KERNEL over the 20 web sites listed in Table 6.3, and measure the number of IPC calls between the untrusted web browser and the TEMI. The last column in Table 6.3 shows that for most

cases, less than 1% of all access to JavaScript data types require crossing the browser-TEMI boundary. Given that the overhead of sending a message via Unix domain socket is around 5 microseconds, a web site with 1,000 IPC messages may only incur 0.005 second overhead.⁵ Thus, the design of separating the execution of CRYPTON functions in the TEMI does not cause any performance bottleneck.

6.3.4 Study on Firefox Vulnerabilities

To estimate the reduction of sensitive data's exposure to browser vulnerabilities, we choose Firefox, which is an open-source and standalone browser with the longest development history. We manually map the 21 high-level Firefox components to our design of the CRYPTON-KERNEL, and estimate the number of security vulnerabilities that could compromise the security guarantees provided by the CRYPTON-KERNEL. We find that all major Firefox components, such as the HTML parser, the JavaScript engine, the Document Object Model (DOM), the network module, etc., belong to the browser code used by the CRYPTON-KERNEL, and are outside our TCB.

We leverage the results in Chapter 4, and categorize historic security vulnerabilities [55] in each component according to 6 severity levels, as shown in Table 4.2. We find that Category 5, i.e., arbitrary code execution vulnerabilities have the largest number. They could allow an attacker to bypass all existing security mechanisms in the browser, and to read or write arbitrary information. If these vulnerabilities fall outside the CRYPTON-KERNEL's TCB, they do not affect the security properties provided by the CRYPTON-KERNEL, as the CRYPTON-KERNEL does not rely on browser code to enforce the properties. However, we find 7 of them roughly map to the functionality of the CRYPTON-KERNEL, such as the server to the CRYPTON-KERNEL communication, and rendering operation processing, etc. These 7 vulnerabilities could potentially compromise our TCB and break the security guarantees provided by the CRYPTON-KERNEL.

20 vulnerabilities in Category 1 break our assumptions, as they may cause mistaken trust on fake SSL certificates, or trick users into making unintended actions. They cannot be addressed by our solution.

The rest of vulnerabilities lead to information leakage or privilege escalation between different web origins, or between web origins and browser chrome. These vulnerabilities cannot affect the TCB of the CRYPTON-KERNEL, and thus are prevented from violating sustained information control guaranteed by our solution. Thus, in total, our solution is not applicable to 27 vulnerabilities, but eliminates the rest of 333 (92.5%) vulnerabilities from compromising the confidentiality and integrity of sensitive web content.

⁵Outliers are Facebook Friend Search and WindowsLive Signup, with high browser-TEMI interactions. A closer look at them reveals that they are caused by a particular CRYPTON function in each of them, which processes both sensitive and non-sensitive information. We separate each function into two versions accordingly, and browser-TEMI interactions fall below 1% among all JavaScript data type accesses.

6.4 Related Work

In this section, we discuss research related to protect sensitive information in web browsers, and key differences with our work.

6.4.1 Enhancement of Browser Security Mechanisms

There has been substantial amount of research on enhancing security mechanisms of web browsers. One direction is to develop flexible and fine-grained access control on the web platform. ESCUDO [82] is a framework for fine-grained access control over JavaScript's access to DOM elements. It adopts the hierarchical protection rings (HPR) as the security model to assign different pieces of JavaScript in the same application with different privileges. JavaScript can only access DOM elements with the same or larger ring numbers, i.e., same or less privileged. JCShadow [124] controls access to general JavaScript object. It divides a JavaScript context into multiple *shadow contexts*. By default, shadow contexts are isolated from each other. Any sharing between them has to be explicitly allowed by access control rules specified by web applications. Zhou et al. [187] propose a similar fine-grained access framework, which also provides mechanisms for automatic policy generation based on identifying third-party scripts and public contents.

Another active area of research aims to improve separation of privileges in browsers through browser redesign [36, 15, 62, 63, 169, 154, 75, 31, 158, 13]. They use various techniques to reduce the impact of untrusted components of browsers and web applications.

A main issue with above solutions is that once sensitive information passes the check, it is no longer under control. The JavaScript will obtain the full information and is free to leak to any party. The web application cannot control how sensitive information can be used, but has to decide either to give up the full information or deny the access. In fact, the assumption of trusting any JavaScript in the web application is ungrounded in practice. Cross-site scripting, malicious browser components or add-ons may tamper with the privileged JavaScript code.

Information flow tracking has also been used for detecting and preventing information leakage in web applications [163]. However, without controlling the usage of sensitive information of client-side JavaScript, malicious scripts can still leak information via self-exfiltration attacks [32]. Moreover, such solutions rely on the trust of browsers.

On the contrary, our solution protects information from unauthorized access using cryptographic techniques. It only disclose the original sensitive information to verified and signed CRYPTON-processing functions. All other code can only process the information by calling these functions. By doing so, we achieve sustained information control all the time. Another advantage of our solution is that we only need to trust a small set of code, while leaving the bulk of browser components outside our TCB.

6.4.2 Cryptographic Techniques

Cryptographic techniques have long been protecting security in different systems. Lie et al. [95] propose an abstract machine of execute-only memory (XOM) to prevent tampering of instructions stored on memory.

A recent work [34] further applies memory encryption to data confidentiality. The idea proposed in the work, self-protecting data, is similar to Data Capsules [100]. It allows unvetted programs to use sensitive data while enforcing policy to confine activities they can perform on the data. The high-level application-specific policy is translated into low-level tags enforced by the underlying hardware. The major weakness with this solution is that leakage via control flows is not considered. This is the consequence of the same fact with access control mechanisms: once the plaintext is disclosed to certain code, it is very hard to control information leakage any further. For example, according to different bits in a secret value, the application code can send a different request to an attacker-controlled server. Our solution maintains the control on potential leakage via control flows by limiting the plain-text only to CRYPTON functions, whose return values are also encrypted by default.

CleanOS [155] uses information flow tracking and encryption to protect mobile devices against threats when the mobile device is lost. Compare to our work, CleanOS mainly focuses on protecting data on mobile devices from device loses, while our work gives stronger security guarantee on a class of sensitive data.

Policy-sealed data [144] is a recent proposal on building trusted cloud services, protecting data from unintended access. Their abstractions and cryptographic primitives (attribute-based encryption) are well suited for the cloud environment. However, unlike on the cloud, where environments can be summarized as configurations, in web browsers, there is no static context or environment. Code from multiple sources with various privileges are running in and molding the mixed browser environment. Therefore, we need more dynamic mechanisms to control data access and processing.

6.5 Summary

In this work, we present a novel abstraction, CRYPTON, which protects sensitive data with memory encryption and allows operations to securely compute over them. We propose a small standalone engine the CRYPTON-KERNEL that is integrated into web browsers to interpret sensitive web content in CRYPTONS. Our large-scale evaluation demonstrates our solution can effectively protect sensitive web content on the web, with reasonable performance.

Chapter 7

Conclusion

In this thesis, we identify the fundamental lack of security support in current web browsers, which is the underlying cause for the overwhelming number of web-based attacks in the wild. To improve browser security for modern web applications, we argue that we need to proactively build well-established security principles into the web browser, such as *separation of privilege*, *least privilege*, *complete mediation*, and *economy of mechanism*. As our solutions, we develop a comprehensive and flexible confinement mechanism for untrusted JavaScript libraries (such as ads), which is a common threat to modern web applications. It isolates the execution of untrusted scripts in a separate environment and regulates all their access to the hosting page. Following our argument for the importance of balancing security-performance trade-offs, we propose a measurement-based methodology to quantify the security and performance implications of privilege-separated browser designs. We apply our methodology to study 9 recent browser designs. To control web application behaviors within a program partition, we propose a general security framework to allow flexible development and deployment of security solutions in the browser. It provides key security support for existing and potentially new security solutions. Lastly, to protect critical pieces of sensitive data on the web, we propose a novel solution to secure their confidentiality and integrity. Our solution provides sustained and complete control on such information throughout its lifecycle, which is not practically achievable via isolation or access control in the complex codebase of browsers. This thesis demonstrates the necessity and practicality of securing web browsers with traditional security principles, which is a promising direction to build a more secure platform for modern web applications.

Bibliography

- [1] Bromium. <http://www.bromium.com/>.
- [2] Invincea. <http://www.invincea.com/>.
- [3] Adobe. PhoneGap. <http://phonegap.com/>.
- [4] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, 2010.
- [5] Devdatta Akhawe, Prateek Saxena, and Dawn Song. Privilege separation in html5 applications. In *Proceedings of the 21st USENIX Security Symposium*, 2012.
- [6] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Security Symposium*, 2010.
- [7] Alexa. Top sites. <http://www.alexa.com/topsites>.
- [8] Anonymous. Summary of source code modification in micro case study. <https://github.com/anonymous-repo/casestudy>.
- [9] Jean Bacon, Ken Moody, and Walt Yao. A model of oasis role-based access control and its support for active security. *ACM Transactions on Information and System Security (TISSEC)*, 2002.
- [10] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS)*, 2010.
- [11] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.
- [12] Adam Barth. Timing attacks on css shaders. <http://www.schemehostport.com/2011/12/timing-attacks-on-css-shaders.html>, 2011.

- [13] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.
- [14] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.
- [15] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The security architecture of the chromium browser. <http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf>.
- [16] Adam Barth, Benjamin I. P. Rubinstein, Mukund Sundararajan, John C. Mitchell, Dawn Song, and Peter L. Bartlett. A learning-based approach to reactive security. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security (FC)*, 2010.
- [17] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin javascript capability leaks: detection, exploitation, and defense. In *Proceedings of the 18th USENIX Security Symposium*, 2009.
- [18] Daniel J. Bernstein. Some thoughts on security after ten years of gmail 1.0. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW)*, 2007.
- [19] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 2001.
- [20] Prithvi Bisht and V. N. Venkatakrisnan. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.
- [21] Bitbucket. <https://bitbucket.org/>.
- [22] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.
- [23] Hristo Bojinov, Daniel Sanchez, Paul Reber, Dan Boneh, and Patrick Lincoln. Neuroscience meets cryptography: designing crypto primitives secure against rubber hose attacks. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

- [24] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
- [25] David Brumley and Dawn Song. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
- [26] Commtouch Cafe. Nasty facebook picture attack based on self-xss - how does this work? <http://blog.commtouch.com/cafe/web-security/nasty-facebook-picture-attack-based-on-self-xss/>.
- [27] CairoGraphics. `cairo_glyph_t`. <http://cairographics.org/manual/cairo-text.html#cairo-glyph-t>.
- [28] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Security Symposium*, 2012.
- [29] Certification Authorities Software Team (CAST). What is a "decision" in application of modified condition/decision coverage (mc/dc) and decision coverage (dc)? http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf.
- [30] 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)*, 2008.
- [31] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.
- [32] Eric Yawei Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *Proceedings of the Workshop of Web 2.0 Security & Privacy (W2SP)*, 2012.
- [33] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.
- [34] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.
- [35] Chromium. Process models - process-per-site-instance. http://www.chromium.org/developers/design-documents/process-models#1_Process_per_Site_Instance.

- [36] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.
- [37] 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)*, 2008.
- [38] Douglas Crockford. ADsafe. <http://www.adsafe.org/>.
- [39] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. CsFire: transparent client-side mitigation of malicious cross-domain requests. In *Proceedings of the 2nd International Conference on Engineering Secure Software and Systems (ESSoS)*, 2010.
- [40] AjaxIM RPG Developers. Ajaxim rpg. <http://ajaximrpg.sourceforge.net/>.
- [41] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.
- [42] Yu Ding, Tao Wei, Tielei Wang, Zhenkai Liang, and Wei Zou. Heap Taichi: Exploiting Memory Allocation Granularity In Heap-Spraying Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.
- [43] Xinshu Dong, Kailas Patil, Xuhui Liu, Jian Mao, and Zhenkai Liang. An extensible security framework in web browsers. Technical Report TR-SEC-2012-01, Systems Security Group, School of Computing, National University of Singapore, February 2012.
- [44] Xinshu Dong, Kailas Patil, Jian Mao, and Zhenkai Liang. A comprehensive client-side behavior model for diagnosing attacks in ajax applications. In *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2013.
- [45] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. Adstentry: comprehensive and flexible confinement of javascript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.
- [46] Nishant Doshi. Please send me your facebook anti-csrf token! <http://www.symantec.com/connect/blogs/please-send-me-your-facebook-anti-csrf-token>.
- [47] DoubleVerify. Doubleverify uncovers ad fraud tied to copyright infringement sites, costing online advertisers \$6.8 million per month.

<http://www.doubleverify.com/resources/research/DV-Fraud-Lab-Report-2013-05/>, May 2013.

- [48] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002.
- [49] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browser against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.
- [50] Facebook. FBJS (Facebook JavaScript). <http://developers.facebook.com/docs/fbjs/>.
- [51] Adrienne Porter Felt, Matthew Finifter, Joel Weinberger, and David Wagner. Diesel: applying privilege separation to database access. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.
- [52] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing capability leaks in secure javascript subsets. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.
- [53] Firebug. Firebug. Web Development Evolved. <http://getfirebug.com/>.
- [54] Internet Engineering Task Force. Rfc 6101: The secure sockets layer (ssl) protocol version 3.0. <http://tools.ietf.org/html/rfc6101>, 2011.
- [55] Mozilla Foundation. Mozilla foundation security advisories. <http://www.mozilla.org/security/announce/>.
- [56] WordPress Foundation. Wordpress. <http://wordpress.org/>.
- [57] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.
- [58] Sujata Garera, Niels Provos, Monica Chew, and Aviel D. Rubin. A framework for detection and measurement of phishing attacks. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM)*, 2007.
- [59] Github. <https://github.com>.
- [60] Goldberg, Wagner, Thomas, and Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 5th USENIX Security Symposium*, 1996.

- [61] Google Inc. Google's income statement information. <http://investor.google.com/financial/tables.html>, 2011.
- [62] 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*, 2008.
- [63] Chris Grier, Shuo Tang, and Samuel T. King. Designing and implementing the op and op2 web browsers. *ACM Transactions on Information and System Security (TISSEC)*, 2011.
- [64] Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th USENIX Security Symposium*, 2009.
- [65] 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)*, 2009.
- [66] Saikat Guha, Bin Cheng, Alexey Reznichenko, Hamed Haddadi, and Paul Francis. Privad: Rearchitecting Online Advertising for Privacy. Technical Report MPI-SWS-2009-004, Max Planck Institute for Software Systems, Germany, 2009.
- [67] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, 2009.
- [68] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005.
- [69] Steve Hanna, Richard Shin, Devdatta Akhawe, Prateek Saxena, Arman Boehm, and Dawn Song. The emperor's new api: On the (in)secure usage of new client side primitives. In *Proceedings of the Workshop of Web 2.0 Security and Privacy (W2SP)*, 2010.
- [70] Robert Hansen and Jeremiah Grossman. Clickjacking. <http://www.sectheory.com/clickjacking.htm>, 2008.
- [71] Thomas E. Hart, Marsha Chechik, and David Lie. Security benchmarking using partial verification. In *Proceedings of USENIX Workshop on Hot Topics in Security (HotSec)*, 2008.
- [72] Mario Heiderich. Towards elimination of xss attacks with a trusted and capability controlled dom. <http://heideri.ch/thesis>.

- [73] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schechter, and Collin Jackson. Clickjacking: attacks and defenses. In *Proceedings of the 21st USENIX Security Symposium*, 2012.
- [74] IBM. Worklight. <http://www-01.ibm.com/software/mobile-solutions/worklight/>.
- [75] IEBlog. Tab isolation. <http://blogs.msdn.com/b/ie/archive/2010/03/04/tab-isolation.aspx>.
- [76] Internet Engineering Task Force (IETF). Rfc 6454: The web origin concept. <http://www.ietf.org/rfc/rfc6454.txt>.
- [77] Intel. Pin - a dynamic binary instrumentation tool. <http://www.pintool.org/>.
- [78] Intel. Trusted compute pools with intel[®] trusted execution technology. <http://www.intel.com/txt>.
- [79] Scott Isaacs and Dragos Manolescu. WebSandbox - Microsoft Live Labs. <http://websandbox.livelabs.com/>, 2009.
- [80] Collin Jackson, Daniel R. Simon, Desney S. Tan, and Adam Barth. An evaluation of extended validation and picture-in-picture phishing attacks. In *Proceedings of the 11th International Conference on Financial Cryptography and 1st International Conference on Usable Security (FC/USEC)*, 2007.
- [81] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.
- [82] Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, 2010.
- [83] 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)*, 2007.
- [84] Martin Johns and Justus Winter. RequestRodeo: Client-side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference, Refereed Papers Track, Report CW448*, 2006.
- [85] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Proceedings of the 2nd IEEE Conference on Security and Privacy in Communications Networks (SecureComm)*, 2006.

- [86] 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)*, 2012.
- [87] Khronos. WebGL specification. <https://www.khronos.org/registry/webgl/specs/1.0/>.
- [88] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, 2006.
- [89] Nadim Kobeissi, Arlo Breault, and Elisabeth Gill. Cryptocat. <https://crypto.cat/>.
- [90] Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium (CSF)*, 2009.
- [91] LastPass. Lastpass password manager. <https://lastpass.com/>.
- [92] Zhichun Li, Tang Yi, Yinzhi Cao, Vaibhav Rastogi, Yan Chen, Bin Liu, and Clint Sbisá. WebShield: Enabling various web defense techniques without client side modifications. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.
- [93] Zhenkai Liang, V.N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, 2003.
- [94] David Lie and M. Satyanarayanan. Quantifying the strength of security systems. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Security (HotSec)*, 2007.
- [95] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.
- [96] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2012.
- [97] Long Lu, Vinod Yegneswaran, Phillip Porras, and Wenke Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

- [98] S. Maffeis, J.C. Mitchell, and A. Taly. Run-time enforcement of secure javascript subsets. In *Proceedings of the Workshop of Web 2.0 Security & Privacy (W2SP)*, 2009.
- [99] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF)*, 2009.
- [100] Petros Maniatis, Devdatta Akhawe, Kevin Fall, Elaine Shi, Stephen McCamant, and Dawn Song. Do you know where your data are?: secure data capsules for deployable data protection. In *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.
- [101] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Proceedings of the 13th International Conference on Financial Cryptography and Data Security (FC)*, 2009.
- [102] Matthew. Facebook’s response to uproar over ads. http://endofweb.co.uk/2009/07/facebook_ads_2/.
- [103] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, 1976.
- [104] Scott McCloud. The Chrome Comic Book, 2008. <http://www.google.com/googlebooks/chrome/index.html>.
- [105] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.
- [106] James Mickens and Mohan Dhawan. Atlantis: robust, extensible execution environments for web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
- [107] MITRE. Common vulnerabilities and exposures. <http://cve.mitre.org/>.
- [108] Mozilla. Bugzilla@mozilla. <https://bugzilla.mozilla.org/>.
- [109] Mozilla. Components.utils.evalInSandbox. <https://developer.mozilla.org/en/Components.utils.evalInSandbox>.
- [110] Mozilla. Dromaeo javascript performance testing. <http://dromaeo.com/>.
- [111] Mozilla. Firefox. <http://www.mozilla.org/en-US/firefox>.
- [112] Mozilla. Firefox OS. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS.

- [113] Mozilla. Same origin policy for javascript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.
- [114] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.
- [115] Ryan Naraine. Research: 1.3 Million Malicious Ads Viewed Daily. http://threatpost.com/en_us/blogs/research-13-million-malicious-ads-viewed-daily-051910.
- [116] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed Systems Security Symposium (NDSS)*, 2005.
- [117] Cong Nie. Dynamic root of trust in trusted computing. http://www.tml.tkk.fi/Publications/C/25/papers/Nie_final.pdf.
- [118] Department of Defense Standard. Department of defense trusted computer system evaluation criteria, 5200.28-std, 1985.
- [119] OWASP. Buffer overflow. https://www.owasp.org/index.php/Buffer_overflow.
- [120] OWASP. Cross-site scripting (xss). [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)).
- [121] OWASP. Integer overflow. https://www.owasp.org/index.php/Integer_overflow.
- [122] OWASP. Time of check, time of use race condition. https://www.owasp.org/index.php/Time_of_check,_time_of_use_race_condition.
- [123] Wladimir Palant. Adblock Plus. <https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/>.
- [124] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards fine-grained access control in javascript contexts. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, 2011.
- [125] 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)*, 2009.

- [126] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADSafety type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX Security Symposium*, 2011.
- [127] The Chromium Projects. Chromium os. <http://www.chromium.org/chromium-os>.
- [128] The Chromium Projects. GPU command buffer. <http://www.chromium.org/developers/design-documents/gpu-command-buffer>.
- [129] The Chromium Projects. Linuxsandboxing. https://code.google.com/p/chromium/wiki/LinuxSandboxing#The_seccomp-bpf_sandbox.
- [130] The Chromium Projects. Sandbox. <http://www.chromium.org/developers/design-documents/sandbox>.
- [131] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
- [132] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iframes point to us. In *Proceedings of the 17th USENIX Security Symposium*, 2008.
- [133] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagnendra Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets*, 2007.
- [134] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the 18th USENIX Security Symposium*, 2009.
- [135] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
- [136] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, 2009.
- [137] Eric Rescorla. Is finding security holes a good idea? *IEEE Security and Privacy*, January 2005.
- [138] Lionel Litty Richard Ta-Min and David Lie. Splitting interfaces: Making trust between applications and operating systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

- [139] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium of Security and Privacy*, 2012.
- [140] Chris Rohlf and Yan Ivnitskiy. Attacking clientside jit compilers. http://www.matasano.com/research/Attacking_Clientside_JIT_Compilers_Paper.pdf.
- [141] RSsnake. XSS (Cross Site Scripting) Cheat Sheet. <http://hackers.org/xss.html>.
- [142] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *The 4th ACM Symposium on Operating System Principles (SOSP)*, 1973.
- [143] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 1996.
- [144] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX Security Symposium*, 2012.
- [145] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.
- [146] Azimuth Security. The chrome sandbox part 2 of 3: The IPC framework. <http://blog.azimuthsecurity.com/2010/08/chrome-sandbox-part-2-of-3-ipc.html>.
- [147] WhiteHat Security. Website security statistics report, May 2013.
- [148] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium*, 2010.
- [149] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.
- [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)*, 2010.
- [151] Symantec. Vulnerability trends, web browser vulnerabilities. http://www.symantec.com/threatreport/topic.jsp?id=vulnerability_trends&aid=web_browser_vulnerabilities.

- [152] Shuo Tang, Nathan Dautenhahn, and Samuel T. King. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.
- [153] Shuo Tang, Chris Grier, Onur Aciicmez, and Samuel T. King. Alhambra: a system for creating, enforcing, and testing browser security policies. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, 2010.
- [154] 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)*, 2010.
- [155] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.
- [156] The Caja Team. Google Caja. <http://code.google.com/p/google-caja/>.
- [157] The RoundCube Team. Roundcube. <http://www.roundcube.net/>.
- [158] 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 Security Symposium*, 2010.
- [159] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.
- [160] Vincent Toubiana, Arvind Narayanan, and Dan Boneh. Adnostic: Privacy Preserving Targeted Advertising. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.
- [161] Minh Tran, Xinshu Dong, Zhenkai Liang, and Xuxian Jiang. Tracking the trackers: Fast and scalable dynamic analysis of web content for privacy violations. In *Proceedings of the 10th International Conference on Applied Cryptography and Network Security (ACNS)*, 2012.
- [162] V. N. Venkatakrishnan, Prithvi Bisht, Mike Ter Louw, Michelle Zhou, Kalpana Gondi, and Karthik Thotta Ganesh. Webapparmor: a framework for robust prevention of attacks on web applications. In *Proceedings of the 6th International Conference on Information Systems Security (ICISS)*, 2010.
- [163] 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 *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

- [164] W3C. The canvas element. <http://www.w3.org/TR/html5/the-canvas-element.html#the-canvas-element>.
- [165] W3C. Cross-origin resource sharing. <http://www.w3.org/TR/cors/>.
- [166] W3C. Document Object Model (DOM) Specifications. <http://www.w3.org/DOM/DOMTR>.
- [167] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.
- [168] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in mashups. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.
- [169] 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 USENIX Security Symposium*, 2009.
- [170] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: improving ssh-style host authentication with multi-path probing. In *Proceedings of USENIX 2008 Annual Technical Conference (ATC)*, 2008.
- [171] Wikipedia. DigiNotar. <http://en.wikipedia.org/wiki/DigiNotar>.
- [172] Wikipedia. Legacy DOM. http://en.wikipedia.org/wiki/Document_Object_Model#Legacy_DOM.
- [173] Wikipedia. Online advertising - Revenue models. http://en.wikipedia.org/wiki/Online_advertising#Revenue_models.
- [174] Wikipedia. Quirk Mode. http://en.wikipedia.org/wiki/Quirk_mode.
- [175] Wikipedia. Tag Soup. http://en.wikipedia.org/wiki/Tag_soup.
- [176] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
- [177] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.
- [178] Zishuang (Eileen) Ye and Sean Smith. Trusted paths for browsers. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

- [179] 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 30th IEEE Symposium on Security and Privacy*, 2009.
- [180] 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)*, 2007.
- [181] Chuan Yue and Haining Wang. Anti-phishing in offense and defense. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC)*, 2008.
- [182] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*, 2009.
- [183] Michal Zalewski. Browser security handbook. <http://code.google.com/p/browsersec/wiki/Main>.
- [184] Stephan Arthur Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.
- [185] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
- [186] Yue Zhang, Jason I. Hong, and Lorrie F. Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, 2007.
- [187] Yuchen Zhou and David Evans. Protecting private web content from embedded scripts. In *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS)*, 2011.