Designing and Implementing the OP and OP2 Web Browsers

CHRIS GRIER, SHUO TANG and SAMUEL T. KING, University of Illinois at Urbana-Champaign

Current web browsers are plagued with vulnerabilities, providing hackers with easy access to computer systems via browser-based attacks. Browser security efforts that retrofit existing browsers have had limited success because the design of modern browsers is fundamentally flawed. To enable more secure web browsing, we design and implement a new browser, called the OP web browser, that attempts to improve the state-of-the-art in browser security. We combine operating system design principles with formal methods to design a more secure web browser by drawing on the expertise of both communities. Our design philosophy is to partition the browser into smaller subsystems and make all communication between subsystems simple and explicit. At the core of our design is a small *browser kernel* that manages the browser subsystems and interposes on all communications between them to enforce our new browser security features.

To show the utility of our browser architecture, we design and implement three novel security features. First, we develop flexible security policies that allow us to include browser plugins within our security framework. Second, we use formal methods to prove useful security properties including user interface invariants and browser security policy. Third, we design and implement a browser-level information-flow tracking system to enable post-mortem analysis of browser-based attacks.

In addition to presenting the OP browser architecture, we discuss the design and implementation of a second version of OP, OP2, that includes features from other secure web browser designs to improve on the overall security and performance of OP. To evaluate our design, we implemented OP2 and tested both performance, memory, and filesystem impact while browsing popular pages. We show that the additional security features in OP and OP2 introduce minimal overhead.

Categories and Subject Descriptors: D.4.6 [**Software**]: Operating Systems—*Security and protection*

General Terms: Design, Security, Verification

Additional Key Words and Phrases: Web browsing, security, browser plugin, formal verification, OP browser

ACM Reference Format:
Grier, C., Tang, S., and King, S. T. 2011. Designing and implementing the OP and OP2 web browsers. ACM Trans. Web 5, 2, Article 11 (May 2011), 35 pages.
DOI = 10.1145/1961659.1961665 http://doi.acm.org/10.1145/1961659.1961665

1. INTRODUCTION

Current web browsers provide attackers with easy access to modern computer systems. According to a recent report by Symantec [Turner 2007], over the last year Internet Explorer had 93 security vulnerabilities, Mozilla browsers had 74 vulnerabilities, Safari

A preliminary version of this article appeared in the *Proceedings of the 2008 IEEE Symposium on Security and Privacy*.
This research was funded in part by a grant from the Internet Research Center (ISRC) of Microsoft Research, NSF grant CNS CT 0834738, grant N0014-09-1-0743 from the Office of Naval Research, AFOSR MURI grant FA9550-09-01-0539, and Intel and Microsoft under the Universal Parallel Computing Research Center.
Authors' address: C. Grier, S. Tang, and S. T. King, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana IL 61801; email: kingst@illinois.edu.

DOI 10.1145/1961659.1961665 http://doi.acm.org/10.1145/1961659.1961665

had 29 vulnerabilities, and Opera had 9 vulnerabilities. In addition to these browser bugs, there were 301 reported vulnerabilities in browser plugins over the same period of time including high-profile bugs in the Java Virtual Machine [AusCERT (@], the Adobe PDF Reader [Petrkov a], the Adobe Flash Player [Adobe b], and Apple's QuickTime [Petrkov b]. Unfortunately, attackers actively exploit these bugs according to several recent reports [Moshchuk et al. 2006; Provos et al. 2007, 2008; Turner 2007; Wang et al. 2006].

Flaws in the design and architecture of today's web browsers allow this trend of exploitation to continue. Modern web browser design continues to support the original model of browser usage where users viewed several different static pages and the browser itself was the application. However, recent web browsers have become a platform for hosting web-based applications, where each distinct page (or set of pages) represents a logically different application, such as an email client, a calender program, an office application, a video client, a news aggregate, etc. The single-application model provides little isolation or security between distinct applications hosted within the same browser, or between different applications aggregated on the same web page. A compromise occurring on any part of the browser, including plugins, results in a total compromise of all web-based applications running within the browser.

Efforts to provide security in this evolved model of web browsing have had limited success. The *same-origin*[1] *policy*– which states that scripts and objects from one origin should only be able to access other scripts and objects from the same origin – is the security policy most browsers try to implement. However, different browsers have varying interpretations of the same-origin policy [Jackson et al. 2006], and the implementation of this principle tends to be error prone due to the complexity of modern browsers [Chen et al. 2007b]. Furthermore, the same-origin policy is too restrictive for use with browser plugins and as a result browser plugin writers have been forced to implement their own ad-hoc security policies [Adobe a; Sun; Microsoft]. Plugin security policies can contradict a browser's overall security policy, and create a configuration nightmare for users since they have to manage each plugin's security settings independently.

Current research efforts to retrofit today's web browsers help to improve security, but fail to address the fundamental design flaws of current web browsers. One project, *MashupOS* [Wang et al. 2007], proposes new abstractions to facilitate improved sharing among multiple principles hosted in the same web page. Another project, *Script Accenting* [Chen et al. 2007b], encrypts scripts from different domains to improve enforcement of the same-origin policy. Both provide scripts with fine-grain isolation within the same web page. However, these mechanisms run within current web browsers (Internet Explorer) and are only as secure as the browser they run within, which currently is not very secure. Sandboxing systems, such as Tahoma [Cox et al. 2006], prevent browsers from making persistent changes to the system and isolate distinct web-based applications using a Virtual Machine Monitor (VMM). This type of persistent-state restriction can be problematic if users legitimately want to store persistent state and allowing users to store and execute downloaded files gives attackers an avenue into the system. Plus, sandboxing at the web-application level can be too coarse grained since it fails to isolate different scripts and objects within the same web application. Combining current fine-grained isolation techniques with sandboxing systems does not provide a complete solution since it would still rely heavily on the underlying browser itself.

[1]An origin is defined as the domain, port, and protocol of a request.

This article describes the design and implementation of the OP[2] web browser that attempts to address the shortcomings of current web browsers to enable secure web browsing. In our design we break the browser into several distinct and isolated components, and we make all interactions between these components explicit. At the heart of our design is a *browser kernel* that manages each of our components and interposes on communications between them. This model provides a clean separation between the implementation of the browser components and the security of the browser, and it allows us to provide strong isolation guarantees and to implement novel security features.

To show the utility of our browser architecture, we design and implement three novel security features. First, we develop novel and flexible security policies that allows us to include plugins within our security framework. Our policy removes the burden of security from plugin writers, and gives plugins the flexibility to use innovative network architectures to deliver content while still maintaining the confidentiality and integrity of our browser, even if attackers compromise the plugin. Second, we use formal methods to prove useful security properties, for example, the address bar displayed within our browser user interface always shows the correct address for the current web page. Third, we design and implement a browser-level information-flow tracking system to enable post-mortem analysis of browser-based attacks. If an attacker is able to compromise our browser, we highlight the subset of total activity that is causally related to the attack, thus allowing users and system administrators to determine easily which web site lead to the compromise and to assess the damage of a successful attack.

The contributions of this article are as follows.

— We present the design and implementation of a new browser architecture that facilitates the development of novel and flexible browser-level security policies.
— We are the first to enforce plugin security policies explicitly from the browser, and we are the first to cope with compromised plugins while still maintaining a high-level of overall browser security.
— We show how operating system principles can be combined with formal methods as a practical methodology for browser design and implementation.
— We are the first to develop techniques for performing post-mortem analysis of browser-based attacks.

This article describes the design and implementation of the OP web browser, which aims to improve the security of web browsing. We address the functions of a secure web browser by identifying three primary goals. First, we should prevent browser-based attacks. Second, although we hope to prevent attacks, inevitably our browser will contain vulnerabilities; in the event of an exploit, the result of a successful compromise should be contained. Finally, even with exploit prevention and containment, attackers may be able to damage infected systems, so we should provide the ability to recover from successful attacks. In Section 2 and Section 3, we describe the design and architecture of the OP web browser. Section 4 shows how the OP web browser is able to implement security policy and apply policy to plugins. We show how the architecture enables the use of formal methods and model-checking in Section 5 and describe the features that enable OP to assist in analyzing browser based attacks in Section 6. Section 7 presents OP2 which has incorporated techniques from three different secure web browsers to provide enhanced functionality. Section 8 presents an

[2]OP comes from Opus Palladianum, which is one technique used in mosaic construction where pieces are cut into irregular fitting shapes.

evaluation of our browser architecture and a qualitative security analysis. Finally, we present related work in Section 9 and conclude in Section 10.

2. THE OP BROWSER DESIGN AND IMPLEMENTATION

In this section, we describe the design of our OP web browser, which attempts to achieve these goals. First, we discuss our threat model and the principles that guide our design, then we discuss our overall architecture, and finally we describe the individual components that make up our browser. In Sections 4, 5, and 6, we describe in detail the specific security features we implement within our browser that achieve our overall security goals.

2.1 Threat Model and Assumptions

The OP web browser was designed to operate under malicious influence. We consider attacks that originate from a web page, potentially targeting any part of the browser. We assume the attacker can have complete control over the content being served to the web browser. A browser compromise could be any sort of attack provided in this manner, with an attack that results in code execution being the most capable form of exploit.

The OP web browser does not address attacks that operate within modern browser security policy, such as cross-site scripting and request forgery. Our goal is to prevent bugs in the web browser application from degrading the overall security of the browser. For example, memory corruption bugs in the JavaScript interpreter and logic errors within a browser plugin are both types of browser bugs that OP attempts to mitigate. Techniques that secure web applications and provide greater protection against cross-site scripting, cross-site request forgery, and other web application attacks are orthogonal to this work and can be used in conjunction with the OP web browser.

We trust the layers upon which OP is built. Namely, we trust the underlying operating system and Java Virtual Machine (JVM) to enforce isolation for our subsystems. As with other current browsers, we trust DNS names for labeling our security contexts. If an attacker compromises any of these entities, the security of our browser is at risk.

2.2 Design Principles

Overall, we embrace both operating system design principles and formal methods techniques in our design. By drawing on the expertise from both communities, we hope to converge on a better and more secure design. Four key principles guide the design for our web browser:

(1) *Simple and Explicit Communication between Components.* Clean separation between functionality and security with explicit interfaces between components reduces the number of paths that can be taken to carry out an action. This makes reasoning about correctness, both manually and automatically, much simpler.
(2) *Strong Isolation between Distinct Browser-Level Components and Defense-in-Depth.* Providing isolation between browser-level components reduces the likelihood of unanticipated and unaudited interactions and allows us to make stronger claims about general security and the specific policies we implement.
(3) *Design Components to Do the Proper Thing, but Monitor Them to Ensure They Adhere to the Design.* Delegating some of the security logic to individual components makes the browser kernel simpler while still providing enough information to verify that the components faithfully execute their design.

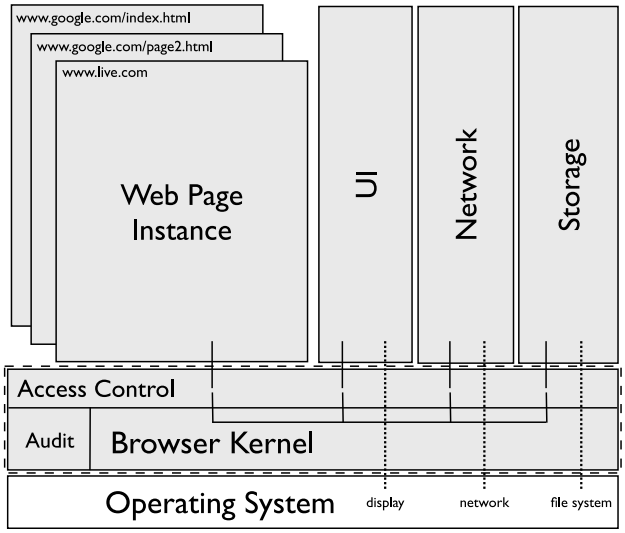

Fig. 1. Overall architecture of our OP web browser. Our web browser contains five main subsystems: browser kernel, storage subsystem, network subsystem, user-interface subsystem, and web page instances; each of these subsystems run within separate OS-level processes.

(4) *Backward Compatibility with Current Web Technologies.* We try to avoid imposing additional burdens on users or web application developers while making the current browsing experience more secure.

3. OP BROWSER ARCHITECTURE

Figure 1 shows the overall architecture of OP. Our browser consists of five main subsystems: the web page subsystem, a network component, a storage component, a user-interface (UI) component, and a browser kernel. Each of these subsystems run within separate OS-level processes, and the web page subsystem is broken into several different processes, shown in Figure 2. The browser kernel manages the communication between each subsystem and between processes, and the browser kernel manages interactions with the underlying operating system.

We use a message-passing interface to support communications between all processes and subsystems. These messages have a semantic meaning (e.g., fetch an HTML document) and are the sole means of communication between different subsystems within our browser. Messages must pass through the browser kernel, and the browser kernel implements our access control mechanism, which can deny any messages that violate our access control policy. We discuss our access control policy in detail in Section 4.

We use OS-level sandboxing techniques to limit the interactions of each subsystem with the underlying operating system. In our current design, we use SELinux [Loscocco and Smalley 2001] to sandbox our subsystems, but other techniques such as AppArmor [Novell 2009], Systrace [Provos 2003], or Janus [Goldberg et al. 1996] would have been suitable for our purposes. Table I summarizes the limitations imposed on each of our subsystems. SELinux is a widely available mandatory access control system for the Linux kernel and many popular Linux distributions include preconfigured policies. Solutions for sandboxing processes also exist in current versions of Windows, Mac OS X, and BSD. In OP, sandboxing strengthens process isolation since

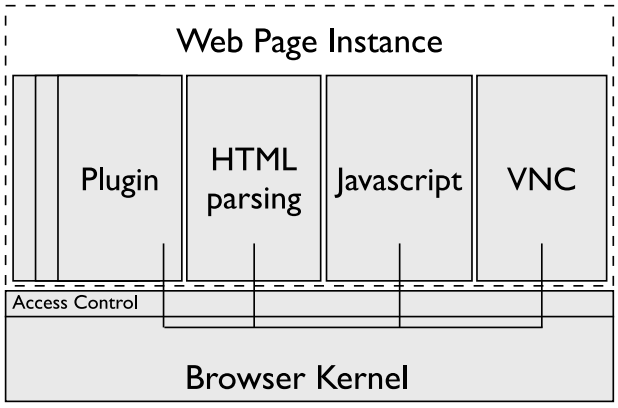

Fig. 2. Within an individual web page instance, each subcomponent runs in a separate OS-level process, and each plugin instance runs within a separate OS-level process. All of the processes communicate through the browser kernel, except for the HTML rendering engine and the plugins, which communicate directly with an Xvnc server. The Xvnc server renders the elements locally and via the browser kernel, streams them to the UI component, where they are displayed for the user. Rendering is the only exception to process communication.

Table I. Summary of OS-Level Sandboxing for Each OP Subsystem

Subsystem	**File system access**	**Network access**
UI	Allowed	Denied
Web page	Denied	Denied
Storage	Limited	Denied
Network	Denied	Allowed
Browser kernel	Allowed	Allowed

all processes are run as the same user and prevents browser exploits from damaging the underlying system.

3.1 The Browser Kernel

The browser kernel is the base of our OP browser, and it has three main responsibilities: to manage subsystems, to manage messages between subsystems, and to maintain a detailed security audit log. To manage subsystems, the browser kernel is responsible for creating and deleting all processes and subsystems. The browser kernel creates most processes when the browser first launches, but it creates web page instances on demand whenever a user visits a new web page. The browser kernel also multiplexes existing web page instances to allow the user to navigate to previous web pages (e.g., the user presses the "back" button).

All messages between subsystems and processes pass through the browser kernel. The browser kernel implements message passing using OS-level pipes, and it maintains a mapping between subsystems and pipes. This mapping allows the browser kernel to avoid source subsystem spoofing, since the browser kernel can accurately identify the subsystem connected to a pipe when it receives a message.

As messages pass through the kernel, they are subject to the access control system. Access control is able to make decisions to allow or deny messages as they pass through the kernel based on the policy implemented. See Figure 3 for a list of messages in OP.

To simplify our implementation, the browser kernel is a single-threaded, event-driven component, and all messages have a unique message ID and a global order.

Source subsystem	Destination subsystem	Message description	Includes return msg
UI, HTML	Kernel	*New web page.* Tells the kernel to open a new web page instance.	No
HTML, JS, Plugin	Network	*Fetch URL.* Fetch an object from the network, return data, redirected URLs, and protocol metadata.	Yes
Kernel	HTML	*Set URL.* Sets the base URL for a newly created web page instance.	No
HTML, Javascript	UI	*Set status, location, title.* Sets the UI status bar, location address, and page title.	No
UI, Xvnc	Xvnc, UI	*Raw VNC data.* Mechanism for transmitting VNC data between the UI and the current Xvnc server. These are two separate one-way messages.	No
UI	Kernel	*Set current web page instance.* Gives the UI the ability to navigate the browsing history.	No
UI	Kernel	*Stop current page loading.* Notifies the kernel that the user wants to stop loading the current web page.	No
Kernel	UI	*New web page instance notification.* Notifies the UI of each new web page instance, the UI uses this information to track browsing history.	No
All	Storage	*Store and retrieve object.* Allows all subsystems to store and retrieve persistant data.	Yes
All	Storage	*Object acl add / remove user.* By default all objects are only accessable by the subsystem that created them, but owning subsystems can add additional readers to stored objects.	No
HTML	UI	*Object ready for download.* Web pages notfiy the UI when downloaded content (e.g., downloaded PDF files) is ready to be saved.	No
Network, HTML	Storage	*Store/retreieve/delete cookies.* Mechanism for the network and HTML (in response to Javascript) to manage cookies.	Yes
HTML, Plugin	Javascript	*Execute Javascript.* Mechanism for executing Javascript.	Yes
HTML	Javascript	*Set Javascript event handler.* Sets the event handling code for Javascript events.	No
HTML	Javascript	*Invoke Javascript event handler.* Invokes the Javascript handling code for a particular event.	Yes
HTML	Plugin	*Set URL.* Sets the base URL for a newly created plugin.	No
Javascript	HTML	*Access DOM element.* Provides access to DOM elements.	Yes
HTML	Plugin	*Call NPAPI function*. The browser makes a call into the plugin	Yes
Plugin	HTML	*Call NPAPI Function.* The plugin makes a call into the browser	Yes

Fig. 3. Message API for OP subsystems. In this figure, we list the origin (or source) of the message and the destination subsystem, as well as a text description explaining the purpose of the message. Some messages include a separate return message that is a reply from the destination subsystem back to the source.

This global order helps make reasoning about security properties easier and reduces many possible race conditions.

The browser kernel maintains a full audit log of all browser interactions. Audit logging is enabled by a database that stores the complete message, including timestamps, headers, and message contents. The browser kernel records *all* messages between subsystems, which enables detailed forensic analysis of our browser if an attacker is able to compromise our system. The audit log operates as write-only storage for the browser kernel. Any log reading access takes place using designated audit log tools separate from the OP browser. This feature ensures that a compromised component does not have access to the browser logs.

3.2 The Web Page Subsystem

Each web page instance represents an individual web page. When a user clicks on a link or is redirected to a new page, the browser kernel creates a new web page instance. For each web page instance, we create a new set of processes to build the web page. Each web page instance consists of an HTML parsing and rendering engine, a JavaScript interpreter, plugins, and an X server for rendering all visual elements

included within the page (Figure 2). The HTML engine represents the root HTML document for the web page instance. The HTML engine delegates all JavaScript interpretation to the JavaScript component, which communicates back with the HTML engine to access any Document Object Model (DOM) elements. We run each plugin object in an OS-level process, and plugin objects also access DOM elements through the HTML engine. All visual elements are rendered in an Xvnc server, which streams the rendered content to the UI component, where it is displayed.

One design decision we make is to use an existing HTML parsing and rendering engine instead of building our own. In our first design iteration, we built our own HTML parsing and rendering engine based on classes provided in Sun's Java runtime. The advantage of this approach is that we could use the type safety properties of Java to provide stronger isolation between individual items within HTML documents (e.g., DOM nodes). However, we found it was difficult to correctly render even simple web pages because of buggy handling of HTML and Cascading Style Sheets (CSS).

To avoid these issues, we use an existing HTML engine called KHTML to parse and render HTML [KDE 2009], and we rely on KHTML to properly parse included-content URLs. The HTML engine tags JavaScript code and browser plugins with the proper source domain. The domains are then used in our security policies to isolate different scripts and objects on the same page. One disadvantage of KHTML as a parsing engine is that it is implemented in an unsafe programming language (C++). To minimize the impact of errors, KHTML can have on the tagging, we still allow KHTML to mark the source domain for JavaScript code and browser plugins, but we check them using our Java-based HTML parser. However, our Java-based HTML parser has difficulty handling today's HTML, so this check produces a silent warning in our audit log rather than halting the web page instance or notifying the user when a violation is detected.

Determining how to include a JavaScript interpreter and plugins was a relatively easy design decision. For JavaScript, we use the Rhino JavaScript interpreter [Mozilla 2009]. Rhino is a high-quality JavaScript interpreter written in Java, which gives us strong isolation between different JavaScript instantiations. Unlike the other components of the web page instance, where functionality is duplicated for each instance, we use a single OS-level process to handle all JavaScript interpretations. We justify this decision by trusting the Java Virtual Machine (JVM) to provide the necessary level of isolation between script objects. For browser plugins we are forced to use existing plugins written in unsafe languages, since there are too many plugins for us to rewrite. However, plugins already have well-defined interactions with the rest of the browser, so we break each plugin instance into a separate OS-level process to provide the necessary level of isolation.

3.3 The User Interface, Network, and Storage Subsystems

Our user interface (UI) subsystem is designed to isolate content that comes from web page instances. The UI is a Java application and implements most typical browser widgets, such as navigation buttons, an address bar, a status bar, menus, and normal window decorations, but it does not render any web page content directly. Instead, the web page instance renders its own content and streams the rendered content to the UI component using the Virtual Network Computing (VNC) protocol [Richardson et al. 1998]. By using Java and having the web page instance render its own content, we enforce isolation and add an extra layer of indirection between potentially malicious content from the network and the content being displayed on the screen. This isolation and indirection allow us to have stronger guarantees that potentially malicious content will not affect the UI in unanticipated ways.

The UI is the only component in our system that has unrestricted access to the underlying file system. Any time the web browser needs to store or retrieve a file, it is accomplished through the UI to make sure the user has an opportunity to validate the action using traditional browser UI mechanisms. This decision is justified, since users need the flexibility to access the file system to download or upload files, but our design reduces the likelihood of a UI subsystem compromise. Providing user interaction for file operation prevents drive-by download attacks as well as arbitrary uploading of files from the user's file system.

Since other components cannot access the file system or the network, we provide components to handle these actions. The storage component stores persistent data, such as cookies, in an SQLite database. SQLite stores all data in a single file and handles many small objects efficiently, making it a good choice for our design, since it is nimble and easy to sandbox. A separate network subsystem implements the HTTP protocol and downloads content on behalf of other components in the system.

4. SECURITY POLICY AND ENFORCEMENT

The OP browser is able to enforce different security policies through flexible access control. We implement three different security policies to explore the access controls that OP offers; in addition to supporting the ubiquitous same-origin policy, we develop two novel policies designed to provide additional flexibility to plugins while still providing the required security for the rest of the browser.

In this section, we discuss the OP security policies. First, we discuss browser plugins and some of the problems with current approaches; then we discuss the policies we implement with a focus on how each policy improves the security of browser plugins. In Section 5, we describe how we use formal methods to show that our access controls are correct and maintain the required security policy in the event of a compromised browser component.

4.1 Browser Plugins

Browser plugins provide the web browser with the ability to view additional types of content. Most web browsers have at least one plugin installed (Adobe reports their Flash Player has been installed on 99% of Internet users' desktops [Adobe 2009b]), and the browser identifies which plugin to use for a particular type of content by the corresponding MIME type. For example, the MIME type "application/x-shockwave-flash" is handled by a Flash-capable movie player such as Adobe Flash Player [Adobe 2009a]. Other popular plugins include Windows Media Player, Adobe Acrobat, Quicktime, RealPlayer, and Java. Plugins provide a variety of functionality, from playing music to just-in-time compilation of programming languages.

Web developers include a plugin within a web page by using the OBJECT or EMBED HTML tags. Figure 4 presents sample HTML used to include plugin content in a web page and shows the specification of MIME types for the included content. The first plugin referenced in Figure 4 includes a Flash movie from YouTube. The Flash movie is executed by the Flash plugin and has the capability to play different video content and interact with the user. The second plugin embeds an instance of a Quicktime-capable player to download and play a video. We refer to the code that is responsible for viewing a particular MIME type as the *plugin* and the content being downloaded and viewed as the *plugin content*.

Plugins complicate browser security because they are given unchecked access to browser internals, making it difficult for the browser to enforce security policies on plugins. Plugins are supplied to the browser in a binary format and usually included as a dynamically loaded library. Though plugins are provided with an application

```
<object width="425" height="355">
  <param name="movie" value="http://www.youtube.com/v/oNsCaVC4Z0o&rel=1"/>
  <param name="wmode" value="transparent"/>

  <embed src="http://www.youtube.com/v/oNsCaVC4Z0o&rel=1"
  type="application/x-shockwave-flash" width="425" height="355"/>
</object>

<object src=
"http://movies.apple.com/movies/paramount/iron_man/iron_man-tlr1_h.640.mov"
type="video/quicktime" width=640 height=288 autoplay="true"/>
```

Fig. 4. This HTML is an page excerpt downloaded from the www.uiuc.edu domain. OBJECT tags are handled by most web clients, while EMBED tags are interpreted by some Mozilla browsers. The URLs that each plugin loads are specified by the SRC attribute. Other parameters for the execution of the plugin can be specified either with PARAM tags or attributes. The first set of tags references Flash content hosted at www.youtube.com, while the second is a movie hosted at movies.apple.com. The HTML also demonstrates how a page can include content from multiple different sources by specifying a URL in the SRC.

programming interface (API) to interact with the browser [Mozilla 2004], plugins run in the same address space as the browser, so the plugin is free to modify browser structures as needed. Thus, a single compromised plugin has complete access to all browser states and events.

Currently, plugin providers implement their own ad-hoc security mechanisms and policies for each different plugin, which causes security problems even for uncompromised plugins. Security policy goals for the browser are not necessarily reflected by the plugin security policy, resulting in inconsistent accesses between the browser and and the plugin. Also, there can be differences in plugin policy between plugin implementations for the same content type. For example, different Flash players could allow different cross-domain accesses based on their developers' interpretation of Flash security policy.

Another aspect of per-plugin security policy is the complicated configuration presented to the user. For example, the Adobe Flash Player provides two different security mechanisms that each require configuration. The first is the plugin's local security settings accessed through an in-browser menu [Adobe a]. The second is a server-side XML manifest governing cross-domain accesses [Adobe 2008]. Since there are a large number of plugins available for modern browsers, requiring the user to configure each one separately is unlikely to be effective.

Providing a common security policy and policy decision point between plugins and the whole browser is important to address both vulnerabilities and malicious plugin usage.

4.2 Plugin Security in OP

To address the shortcomings of current plugins, we design and implement a plugin architecture to provide security for plugins in the OP browser. The OP browser enforces security policy in the browser kernel. Consistent with all other policy decisions, any plugin-related access control is done by the same security mechanisms enforcing policy for the rest of the browser.

To enforce security policy, we interpose on message passing in the browser kernel. Each browser process is labeled with a security context (i.e., domain) depending on the security policy being used. We run each instance of a plugin in a separate process that is assigned its own label by the kernel. In order to correctly label each plugin process, the browser kernel inspects messages that trigger the plugin to load content from a

URL. This security label is then used to make decisions for other plugin and browser actions. The plugin can be denied access to browser resources; similarly, the rest of the browser can be denied access to plugin resources. Each pairwise communication channel between browser subcomponents can have an access control module operate on the messages. Any security-related state is maintained inside the corresponding access control module. Our implementation provides a simple API for constructing different security policies.

4.3 Plugin Security Policies

In addition to developing a plugin architecture, we develop two novel security policies that specifically address the needs of plugin security while still providing enough flexibility to support common plugin usage.

4.3.1 Provider Domain Policy. Provider domain policy allows a plugin embedded in a page to have permissions associated with the source of the plugin content. Media sharing from sites such as YouTube allow one site to host the video content and other content publishers to embed the video into their sites or blogs. Advertisements are provided by an advertising company and similarly embedded along with web page content. In both cases, the web page creator has little control over the content inside an embedded area, especially if it includes plugin content. Our policy is designed to reflect the intent that a web page creator has when embedding videos and content across domain boundaries.

Provider domain policy sets the origin of the plugin to the site hosting the plugin content. When a page uses the OBJECT tag to include a file associated with a plugin, the plugin content is given permissions according to the domain in the OBJECT tag SRC attribute. If same-origin policy is applied to plugins, the browser would associate the plugin content with domain from the URL of the page containing the OBJECT tag. The same HTML from Figure 4 can help to illustrate the difference. Same-origin policy would treat both plugins as if they came from the www.uiuc.edu domain, since that is where the web page is hosted containing the HTML. Our provider domain policy labels the two plugins differently. The first movie would be tagged with the domain www.youtube.com, and the second with movies.apple.com. The page hosting the content is given permissions according to the www.uiuc.edu domain. Label differences force separation between content and prohibit the embedded content from altering the page or fetching any resources associated with the page's domain. Each of the plugins embedded inside the page can access data associated with the corresponding domains. For example, the YouTube video can access cookies, make network connections, and use other resources, but only from www.youtube.com. This example shows how popular use of plugins can be accomplished inside the constraints of browser security policy.

Provider domain policy has several important security implications. As content is shared between sites and included in websites, the page authors do not need to be concerned with security when including cross-domain content. Included content is isolated from the page's domain, and users viewing websites that include cross-domain plugin content are safe from malicious and vulnerable plugins. This policy limits plugins included across domains and prohibits plugin content included in this way from accessing cookies, DOM, and other browser components. Plugin content designed to function like a library for web page authors will have limited functionality under provider domain policy due to these restrictions.

4.3.2 Plugin Freedom Policy. The plugin freedom policy provides additional flexibility to plugins by allowing outgoing network accesses while limiting access to page and user information contained in the browser. This policy is motivated by plugins such

as SopCast [SopCast 2009], a peer-to-peer video player, that need additional flexibility for outgoing network connections.

Plugin freedom policy provides local plugin storage and unlimited network access at the cost of the ability to access DOM and other browser components. To implement this policy using the OP browser access controls, we simply prohibit communication, after the plugin starts up, with any browser components besides the network and storage subsystem. The storage subsystem provides a location for per-plugin storage that is allowed by this policy. Per-plugin storage allows plugins to have access to local settings and save files in a safe environment. All network communications are also allowed. The rest of the browser subcomponents are able to interact as normal and provided with standard same-origin protections.

Plugin freedom policy would prevent some plugin content from functioning properly, though media sites such as YouTube, Apple Movie Trailers, and others continue to operate as desired. The plugin content on these pages does not need to access any of the other browser components besides network resources. Any plugins that interact with the content on pages will not function correctly, since any attempted JavaScript execution will be prevented by the access controls.

This policy is similar to current plugin operation in browsers, except that the scriptable plugin API components are disabled. Removing interaction capability with other browser components prevents plugins from leaking client information across sites if the plugin is exploited or the plugin content is malicious.

5. FORMAL VERIFICATION

We designed the OP web browser to include the use of formal methods to verify its correctness. To better support formal methods, we use small, simple, and exposed APIs that allow us to model our system and reason about it. Using formal methods, we are able to provide greater assurance that we preserve our security goals during an attack and compromise.

We formulate the OP web browser within the logical framework of rewriting logic and use formal reasoning tools to verify model correctness, including the presence of attacks, successful compromises, and access control [Clavel et al. 2002]. The reasoning engine we use is the Maude system [Meseguer 1992]. We use the term “Maude” to refer to both the Maude interpreter and the language.

Once the browser model has been formally specified, we can use Maude’s search ability for model-checking to verify invariants over the finite state space we need to consider. The invariants of a browser system fall into two categories: program invariants and visual invariants. Program invariants for OP consist of the goals of the access control policy. These invariants are relatively easily gathered from the source code and concise specification of security policy. The visual invariants (e.g., preventing address bar spoofing) need extra effort to be mapped into program invariants. In this article, we model these invariants, and we also translate browser compromise and built-in defenses into rewriting logic rules. As we explain in the following, OP’s address bar logic and same-origin policy are specified by rewrite rules and equations in Maude, and we use model-checking to search for spoofing and violation of same-origin policy scenarios. The result of the search is a list of states that are violations of the invariants specified and the sequences of actions that lead to the invalid state. States that are violations of security invariants can assist in the development process by catching potential problems before they are exploited.

In this section, we discuss how we use formal methods to improve the design of our browser. First we give a very brief overview of Maude; Then, we discuss how we created our model. Finally, we describe how we model-check to prove the

```
1 mod SIMPLE-CLOCK is
2   protecting INT .
3   sort Clock .
4   op clock : Int -> Clock [ctor] .
5   var T : Int .
6   rl clock(T) => clock((T+1) rem 24) .
7 endm
```

Fig. 5. A simple Maude example from the Maude Manual (Version 2.3). This example describes a model for a 24-hour clock in Maude.

```
search in SIMPLE-CLOCK :
clock(0) =>* clock(T)
such that T < 0 or T >= 24 .
```

Fig. 6. The search statement from the Maude Manual (Version 2.3) showing how to model-check the SIMPLE-CLOCK model invariant using Maude's search functionality.

absence of address bar spoofing attacks and to verify parts of our same-origin policy model.

5.1 Modeling Using Maude

A simple example using Maude and model-checking of invariants is presented in the Maude Manual [Clavel et al. 2007]. The example involves the model of a clock and uses Maude to search the state space for states with invalid hour values. The Maude model for the clock example is presented in Figure 5. This example illustrates several Maude features, though we only describe the ones relevant to the OP browser model we present later.

Figure 5 shows the Maude model named SIMPLE-CLOCK. The third line defines a sort, called `Clock`. A sort is similar to the `class` keyword in C++ and simply defines a category for later use. Line 4 in the figure contains the definition for an operation called `clock`; operations act on a sort and generally connect a sort (or set of sorts) to a different set of sorts. In the SIMPLE-CLOCK model, we connect the sort `Int` to the sort `Clock`. Operations do not define how Maude connects the two sorts; instead, they only specify the connection. Rewrite laws begin with `rl` and describe transitions between states. The SIMPLE-CLOCK model has one rewrite law. This rewrite law says that the clock operation increments the clock variable T and then takes the remainder after dividing by 24.

Once we define a model in Maude, we can use the `search` function to have Maude explore the state space and find states that match our search criteria. For the SIMPLE-CLOCK example we want to find states that violate an invariant, such as the clock's state being outside of the 0 to 24 range. Figure 6 shows the example search statement for the SIMPLE-CLOCK model. This search statement defines an initial state, 0, and the condition to match when searching.

The Maude model for the OP browser consists mostly of definitions of types and state for each component by defining sorts, operations, and variables for each browser component. To define browser behavior, we use rewrite laws to show transitions between different internal states in the browser. Our invariants are specified as statements and we use the same search functionality in Maude to find matching states.

5.2 Formal Models and System Implementations

There is often a gap between the formal model used to verify properties and the system implementation. While we recognize that this gap exists between our model and

```
<UI-ID : Frame | addrBar: URL, ... >
imsg(count, src, dst, IDENTIFIER, content)
< ... > ...
```

Fig. 7. The message specification in Maude. The first section of the specification is a class-like structure, starting with < and ending in >. UI-ID is the instance identifier of the type, Frame is the type, and after the pipe are the members of the type. The next line begins with `imsg` and is the constructor for the message type. The constructor takes the elements in parentheses and creates an object of a specific type. The `imsg` constructor creates an object of type Message.

system, we feel that for our uses of formal methods the difference is small enough that we are able to use the results of model-checking to iterate on design and development. Since we implement each of the browser components separately and use a compact API for message passing, the model that we use to formally verify parts of our browser is very similar to the actual implementation. The model we create is focused on message passing between components. We do *not* verify, for example, that the HTML parsing engine is bug-free; instead we verify that, even if the HTML parsing engine had a bug, the messages that a code execution attack could generate (potentially any message) would not force the browser as a whole into a bad state. To do this, each component is modeled in Maude and aspects of every component's internal state are included. Messages are the means for the browser's internal state to change.

Our application of formal methods helped us find bugs in our initial implementation. By model-checking our address bar model, we revealed a state that violated our specification of one address-bar visual invariant. The resulting state was actually due to a bug in our implementation, as we had not properly considered the impact of attackers dropping messages or a compromised component choosing to not send a particular message. Our model gives an attacker complete control over the compromised component, including the ability to selectively send some types of messages and not others. We used the result to fix our access control implementation, and we updated our model accordingly.

In the interest of space, we have not included the entire Maude model. In the following sections, we highlight parts of our model that we use to model-check same-origin and visual invariants. We have not specified all browser invariants in our model, as this is a first step in our venture into formally verifying an entire web browser.

5.3 Modeling the OP Browser

Component-based systems can be modeled in Maude as multi-sets of entities, loosely coupled by a suitable communication mechanism. For OP, the entities are browser components, each with a unique identity, and the communication mechanism is the message-passing API. In the Maude version of our OP implementation, the states of OP are represented by symbolic expressions, and the state transitions are specified by rewrite rules describing the components' communication with each other and the state transformation. In this section, we discuss our model for message passing and processing, user actions, and how include browser compromise into this model.

Communication between components in OP is done through the message-passing interface, which is the communication mechanism modeled in Maude. The messages are expressed as entities in the multi-set of components. The message specification in Maude is shown in Figure 7. The messages are tagged with a count to make sure they are processed in the right order. Message ordering is preserved by the browser kernel, and in order to have ordering in the multi-set representation in Maude, a count attribute is introduced. A simple example illustrating our model of the message-passing interface and a corresponding state change is shown in Figure 8. This rule is responsible for updating the browser state, including the address bar of the user interface.

```
< UI-ID : Frame | addrBar : URL, ... >
< MSG-ID : MsgCount | msg-to-process : N,  msg-to-send : M >
imsg(N, webAppId, UI-ID, MSG-SET-LOCATION-BAR, new-URL)
=>
< UI-ID : Frame | addrBar : new-URL, ... >
< MSG-ID : MsgCount | msg-to-process : s(N), msg-to-send : M >
```

Fig. 8. The Maude rule corresponding to the state change due to a SET-LOCATION-BAR message being received. Notation here is similar to that of Figure 7; The first three lines are the current state and creation of the message to be processed. The remaining lines represent the state after the state change. The full browser state includes other components besides the UI and message queue.

```
< UI-ID : Frame | addrBar: URL, ... > GO
< MSG-ID : MsgCount | msg-to-process : N, msg-to-send : M >
=>
< UI-ID : Frame | addrBar : URL, ... >
imsg(M, UI-ID, KERNEL-ID, MSG-NEW-URL, URL)
< MSG-ID : MsgCount | msg-to-process : N, msg-to-send : s(M) >
```

Fig. 9. Maude expression for the "GO" UI button causing a message to be sent. The first line represents the portion of the browser state for the Frame and the user action being performed, which in turn causes produces a new Frame state and the message with type set to MSG-NEW-URL.

The browser state as a whole is represented by the objects corresponding to each of the components. This means that Maude represents a state as a grouping of the UI, network, plugin, and other subsystem states. Figure 8 shows an action that sets the location bar in the UI. The first three lines of Figure 8 are the current browser state and include the creation of a message called MSG-SET-LOCATION-BAR using the `imsg` constructor. The browser state is rewritten, including in the UI a new address in the address bar (shown by new-URL), and the results of the rewrite are the last two lines of the figure. Rewrite rules such as these cause the Maude model to change state. Model-checking through `search` locates possible states that can occur as a result of these rules and satisfy an additional expression.

5.3.1 Modeling User Actions. We also model the user actions in the browser system, such as clicking the "GO" button to request a new web page. The Maude model of the UI is very similar to the Java source code we wrote to implement the UI in the OP web browser. The Maude rule describing the message generation as a result of the "GO" button being clicked is listed in Figure 9. This Maude rule is especially descriptive of the original Java source; as we can see, the message created has the source set to UI-ID, a destination of KERNEL-ID, the message type of MSG-NEW-URL, and the URL that is the content of the message. This is precisely the same set of actions as in the Java that implements the sending of the NEW-URL message. The first two lines of Figure 9 are the current UI and message queue state, plus the user action labeled "GO." The three lines following the => marker are the new browser state, which include a new message being generated by the `imsg` constructor.

5.3.2 Modeling Browser Component Compromise. Our model also includes potential attack paths. As an example of a component-level compromise, the attacker could take control of a web page subsystem instance and, using the message API, force the compromised component to send incorrect URL information to the UI component, resulting in address bar spoofing. Setting the address bar to a different location than the page contents is primarily useful for phishing attacks. Using access controls, we prevent such attacks from being successful. In Maude, we express the compromise of a

```
< UI-ID : Frame | AddrBar : S1:String, NavWebApp : WebApp1:Int,...>
< WebApp2:Int : WebApp | Content : S2:String, ... >
such that (WebApp1:Int == WebApp2:Int) /\ (S1:String =/= S2:String)
```

Fig. 10. A Maude expression describing the condition checked for address bar spoofing. This condition is used as a test for bad browser states. The first line is the current state of the browser, specifying the UI and ID for an instance of the web page subsystem. The last line is the comparison, which checks that the URLs associated with the address bar and web page subsystem are different, indicating a state where the address bar is spoofed.

component as additional rules that generate messages and trigger message passing and processing like ordinary rules.

5.4 Model-Checking Address Bar Invariants

Cases that allow the address bar in the browser to mismatch the page content were examined for Internet Explorer in work by [Chen et al. 2007a]. They searched for violations of invariants specified for GUI elements in Internet Explorer under normal operation. We are able to verify a similar result for the OP browser using our formal model of the message-passing interface and our security policy. The key difference in our approach is that our proof holds even in the presence of a fully compromised web page instance.

To model-check and find cases of address bar spoofing, we must define a good browser state. Once we have an expression for good browser states, we can use Maude to search for the bad ones. We define a good state as a state where the content of the currently navigated web page matches with the URL shown in the address bar. The Maude expression describing spoofing is shown in Figure 10. When we use the model-checking search tool to search from an initial state, consisting of all the components of OP and some user actions, the results show there is no logic error leading to the address bar spoofing scenarios. We also make sure that the address bar cannot be spoofed once the web page subsystem is compromised, showing that the access control logic can defend against possible attack sequences. This result verifies that if the browser kernel and UI are trusted, no sequence of messages can violate our address bar invariant, even if an attacker compromises a web page instance. This result does not provide guarantees for the display of the web page content. For example, a compromised rendering engine could display incorrect content. Our model checking gives us high assurance that such an exploit will not be able to affect the address bar or other UI elements and remain contained inside the rendering engine.

5.5 Model-Checking the Same-Origin Policy

Our implementation of the same-origin policy for the OP web browser controls access to all browser components. We use model-checking to verify that the same-origin policy cannot be violated by a single component being compromised. Although our model focuses on interactions with plugins, other components with similar interactive capabilities can benefit from the result. We model a compromised web page subsystem and plugin, and verify that the access control implemented in the browser kernel enforces the same-origin policy specified as invariants in our model.

Plugins and JavaScript are able to interact with each other through the scriptable plugin extension to the Netscape Plugin API, and we support such interaction in OP. Enforcing the same-origin policy for these components is done in the same manner as our other security policies for plugins in the browser kernel. The simple message API keeps the state space small enough for model-checking to be tractable when considering all the possible actions by different browser components. We prove

a few different invariants. For example, we prove that a plugin from one domain cannot send a message to a plugin (or web page instance) from another domain, and vice-versa.

Introducing binary compatibility for the Netscape Plugin API would increase the size of the message API for plugins, although our access controls still remain in the browser kernel. Once OP is binary compatible with the Netscape Plugin API, we should be able to adapt the model and verify that the same-origin policy is upheld with the added complexity.

6. ANALYZING BROWSER-BASED ATTACKS

Although we put significant effort into securing our OP web browser, attacks may still occur. One class of attacks that may occur are "social engineering" attacks where a user is fooled into performing an action that violates the security of the system. For example, researchers from Google found that attackers fool users into downloading and executing malicious content from adult web sites by making them think they are installing a new video codec in an attempt to view "free" videos [Provos et al. 2007]. A second class of attacks that may occur are web-based application bugs, such as cross-site request forgery attacks [Jovanovic et al. 2006]. In these attacks, a malicious web site can coax the browser into performing the actions of a site, such as transferring money using a banking application, without the users knowledge or consent. The problem with these classes of attacks is that they adhere to the browsers security policy and from the browsers perspective appear to be legitimate actions, making these types of attacks difficult to prevent.

Our goal is to allow users and system administrator to recreate the past to analyze browser-based attacks. To analyze an attack, users and administrators may want to perform two types of analysis. First, they may want to determine which web site initiated the attack so they can blacklist the web site and avoid visiting it again in the future. Second, they may want to track the effects of known-malicious web sites that they visited to determine if they were attacked, or to assess the damage of a successful attack.

One difficulty in analyzing browser-based attacks is that the activities of the attacker are intermingled with legitimate actions. Even if users and system administrators have a complete security audit log that can recreate arbitrary past states and events, most of the content in the audit log is from legitimate web usage. Thus, it is difficult to highlight the subset of browsing activity that is most likely to be part of the attack.

We designed our OP web browser to overcome these shortcomings to enable users and system administrators to better understand browser-based attacks. To highlight the activities of an attacker, we use browser-level dependency graphs to help visualize the attack. Browser-level dependency graphs are graphs of browser-level objects connected by causal events within our browser. A causal event is defined as any event where information flows from one object to another, thus forming a dependency from the source object to the sink object. For example, if a user clicks on a link, this forms a causal link from the web page instance that hosts the link to the new web page instance that the browser kernel creates in response. Using these connections, we generate dependency graphs of attacks to show where an attack came from and to show what effects an attack had on our system.

To give a more concrete example of a dependency graph, Figure 11 shows a graph for a successful browser-based attack from a real web site. For this graph we assume that we (as the user) know the web site videozfree.com is malicious (as was pointed out in a recent paper from Google [Provos et al. 2007]) we want to check if we downloaded

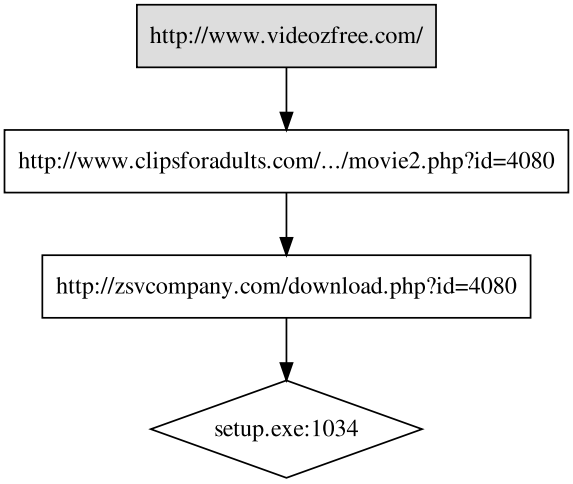

Fig. 11. Forward dependency graph for *videozfree* attack.

files from the site into our file system anytime in the past. Our analysis starts with the videozfree.com web page. From videozfree.com, we clicked on a still image of a video and were sent to clipsforadults.com, which displayed an image that appeared to be a video plugin. When we clicked on the "play" button, it automatically prompted us to download a new codec to view the video, which we downloaded as the file setup.exe. For this experiment, the attack was in the middle of several weeks worth of typical browsing, and our audit log contained 349,313 events and 1218 different web page instances, yet we were able to automatically extract this much smaller subset of the total information available. Also, even though videozfree.com was listed as the malicious web site, the actual download came from a different site (*zsvcompany.com*), and based on monitoring videozfree.com for several months this download site changes periodically making it hard to track using blacklists.

In this section, we discuss our techniques for analyzing browser-based attacks. First we describe the objects we track, the dependency forming events that connect objects, and the dependency graphs we generate to facilitate analysis. Then, we describe an example that illustrates how to analyze a cross-site request forgery attack.

6.1 Intrusion Analysis Design

To track attacks using browser-level dependencies, we need to define the objects we monitor and the events that connect these objects. When defining objects and events we have three main design considerations. First, we must decide at what level of granularity we should display our dependency graph. More coarse-grained dependency graphs will be smaller and easier to analyze manually, but may lack the fidelity to provide useful information about the attack. Second, we must decide at what level of granularity to track dependencies. In general, more coarse-grained dependency tracking will be more efficient to implement, but could lead to false dependencies due to excessive tainting. Third, we must decide which events give the attacker the most direct control over the system and focus our analysis on these events. For example, we do not track the event where a web page instance sets the status bar in the UI. Certainly an attacker could use this as part of an attack, but is it likely to assist the attacker in fooling the user to do something else – such as visit a malicious web site – that we do track.

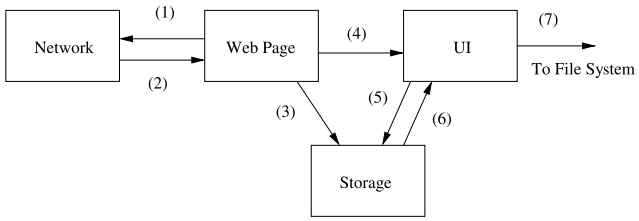

Fig. 12. This figure shows how our browser subsystems interact to download a file to the file system. First, the user clicks on a link which causes (1) the web page to request the file and (2) the network subsystem returns the downloaded content to the web page. Then (3) the web page stores the file in the storage manager and (4) notifies the UI that there is a downloaded file waiting. The UI then prompts the user and (5) retrieves the file from the storage manager (6). Finally, (7) the UI saves the file in the file system.

To strike a balance between analysis fidelity and the amount of information displayed to the user, we track two primary objects: web pages and files. Web page objects map directly to the web page instance subsystem within our browser. Web page objects are identified by the URL used to open the web page and we consider different web page instances with the same base URL to be different objects. Web page objects consist of HTML documents, images, JavaScript, plugins, etc. and even though we do record the interactions between these entities we chose to display them all as a single object to reduce the number of visual elements in our dependency graphs. File objects are file system objects, and we track these as they flow through our browser.

We track events that connect web pages together, and web pages with files. Whenever a web page creates a new web page (e.g., the user clicks on a link), the new web page depends on the web page that initiated the action. When a user downloads or uploads a file, we connect the file with the associated web page.

Although we designed our browser to make dependency-causing events explicit, we still have to put in effort to achieve the fine-grained dependency tracking to support the events we follow. For example, when a user downloads a file and stores it in the file system the downloaded file travels through many different subsystems (Figure 12), but we want to make the connection directly between the web page and the file without including the intermediate subsystems. When a user downloads a file they usually initiate this action by clicking on a link in a web page. The web page then makes a network request and then stores the file as a persistent object in our storage subsystem. Then, the web page notifies the UI that a downloaded file is waiting the storage subsystem, and the UI retrieves the file from the storage subsystem and saves it in the file system. We designed the storage and UI subsystems so that this data will pass through them without being modified, but an attacker that compromises one of these components could violate this assumption and result in dependencies that we miss since the attacker was able to affect data without using an explicit message. To prevent this type of unaudited dependency, we monitor the storage and UI subsystems to verify that all objects and files that pass through them are unmodified. This simple invariant allows us to track these objects at a fine granularity without monitoring the internals of each subsystem. An attacker can still leak information using covert channels [Lampson 1973], but this would require the attacker to compromise two components, not just one. In our current implementation we do not perform this check for the network subsystem, but adding the additional invariant to verify network object integrity would be straightforward.

We apply the BackTracker [King and Chen 2003] graph generation algorithm to create dependency graphs. The BackTracker graph generation algorithm starts with a single object, called a *detection point* and traverses the audit log to find the set of

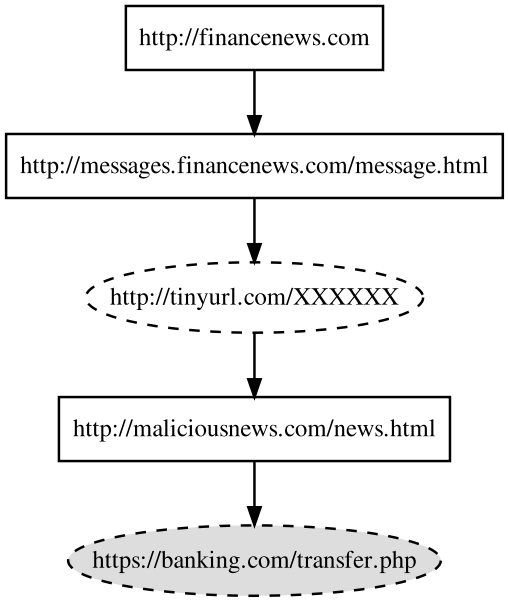

Fig. 13. Backward dependency graph for *xsrf* attack.

objects that are causally connected to the detection point. We use the backward graph generation algorithm [King and Chen 2003] to find the origin of an attack (e.g., malicious web site) and the forward graph generation algorithm [King et al. 2005] to track the effects of an attack.

6.2 Example: Cross-Site Request Forgery

In this section, we discuss an artificial cross-site request forgery attack based on a real attack described by [Stamos and Lackey 2006], in their Black Hat presentation. We show how our analysis techniques can help users understand this type of attack. Our example starts with a victim receiving a monthly banking statement and noticing a spurious transfer of $5000 and our goal is to figure out how this transfer occurred.

The starting point for our analysis is the specific HTTP request that resulted in the transfer of funds. We assume the victim can identify the specific network request that lead to the transfer (perhaps with the help of the bank). Starting with this request, we work our way backward to figure out what went wrong.

Figure 13 shows the dependency graph for this attack. The request in question originated from the maliciousnews.com site. The victim arrived at the maliciousnews.com site by visiting a financial new site, financenews.com and going to message boards (*messages.financenews.com*) to look for "leaked news" about stocks of interest. The victim found a story they were interested in and clicked on a *tinyurl.com* link that redirected them to maliciousnews.com. The news story on the maliciousnews.com site contained an invisible inline HTML frame (iframe) that issued the request to transfer funds. This action was possible because of a design flaw in a banking application that made available to the iframe a login cookie that was created when the user logged into the banking application in a previous browsing session. This attack was *not* the result of a browser bug, the browser correctly enforced the same origin policy in this

Table II. Brief Summary of Differences Between Browser Architectures

Browser	Kernel	Nav.	Sub-windows	Frames	Display policy	Display mech.
OP	microkernel	isolated	isolated	not isolated	none	streaming image
Chrome	monolithic	isolated	different-site isolated	not isolated	none	custom
Gazelle	monolithic	different-origin isolated	isolated	different-origin isolated	opaque overlay	streaming image
OP2	microkernel	isolated	isolate	different-origin isolated	delegate once	window manager

The table specifically identifies differences in architectures between OP, Chrome, Gazelle and OP2.

example. This type of attack is commonly referred to as a cross-site request forgery (xsrf) attack.

One omission from this dependency graph is the cookie connecting the malicious-news.com site with the banking application. We filter out cookies because many web sites use cookies to track users across multiple unrelated sites, and including cookies results in large dependency graphs. We could have created a list of well-known tracking cookies and removed only these tracking cookies from our graphs, but we did not explore this alternative technique for this article.

7. OP2: IMPROVING THE OP WEB BROWSER

After the design and implementation of OP there has been significant effort by both the industry and research communities to design secure web browsers. After OP, two additional web browsers were designed to improve the state-of-the-art in browser security: Google Chrome and Gazelle [Google 2009; Wang et al. 2009]. OP, Chrome, and Gazelle share many common features and each browser was designed from the ground up with security goals in mind. After carefully considering the design choices made by all three browsers, we have designed and implemented a second version of OP, called OP2, that combines techniques from OP, Chrome, and Gazelle. This section discusses the features we have incorporated into OP2 to achieve strong security, adequate performance, and compatibility (see Table II for a summary).

Our goals in the design of OP2 are:

— *Security*. By drawing on the contributions of all three browsers, we hope to achieve a more secure web browser.
— *Performance*. Architectural decisions can impact the speed of the web browser, the browser architecture should have low overhead compared to other, modern web browsers.
— *Compatibility*. Browser design should have a minimal impact on the compatibility of the browser with today's web pages.
— *Few Lines of Code*. OP2 should require as few lines of code as possible to keep the design simple and easy to reason about. A small code base also enables it to be used by the research community and simplifies maintenance.

7.1 Browser Kernel Architecture

OP has a minimalistic browser kernel design that facilitates message passing between browser processes. The browser kernel is a microkernel by design and has dedicated

processes that provide access to system resources. In addition to message passing, the browser kernel uses an access control policy to restrict messages between browser processes and implements all of the security policy for OP.

Gazelle and Chrome both have a monolithic browser kernel that provides services, such as network and cookie access, on behalf of the rendering process. The browser kernel in Gazelle is around 5,000 lines of C# code and uses the .NET framework and the Chrome kernel uses over 110,000 lines of C++ code (the code count is from the Linux version of chromium r22885 and excludes common libraries and third-party code). In contrast, the browser kernel in OP is around 1,200 lines of C++ and uses only the standard libraries.

OP2 uses the original OP design for the browser kernel because the microkernel approach simplifies our implementation of the browser kernel while still providing the necessary information for our access control policies. We have found that the additional latency from message passing required in a microkernel design adds very little overhead because of added parallelism, as shown in Section 8. We do perform some simple optimizations in the browser kernel to enhance the speed of OP2, which we discuss in Section 7.6.

7.2 Process Models

Before we discuss the different process models supported by OP, Chrome, and Gazelle, we first establish some terminology. To be consistent with the prior description of OP, we call all of the processes responsible for executing content in a single page the *web page instance*, Figure 1 shows the logical grouping of processes into a web page instance in OP2. The web page instance is called the *rendering instance* in Chrome, and the *principal instance* by Gazelle [Google 2009; Wang et al. 2009]. All three browsers architectures have similar OS based designs: a browser kernel manages one or more web page instances. Web page instances have two dimensions of potential reuse: during navigation from one page to the next and between tabs or windows. Additionally, each of the browsers has made slightly different decisions on how to create web page instances when a page creates new windows with JavaScript.

The standard operation of OP uses a different web page instance for each page. In OP, each web page instance consists of a JavaScript, HTML, VNC, and multiple plugin processes as described in Section 3. When the user navigates between pages, using a link on the page or by entering a URL into the address bar, OP uses a new web page instance for the new page. OP also uses a web page instance for each tab. If the page attempts to create a new window, that window is an independent web page instance.

Chrome, like OP, uses a web page instance for each page in addition to supporting other models. Chrome calls the default process model *process-per-site-instance* and supports other process models as command line arguments. The default mode in Chrome is nearly identical to the OP model, except for one small difference. In Chrome, when a new window is created by JavaScript in another page, it is run in the same web page instance if the registered domain name and protocol match (i.e., http://docs.google.com is the same as http://www.google.com). Additionally, same-origin navigation does not cause the creation of a new web page instance. Chrome uses a single process for the web page instance, combining what OP uses three processes to accomplish. Reis and Gribble [2009] discuss the Chrome process models in greater detail; different tradeoffs for each process model are also discussed in the developer documentation [Google 2008].

Gazelle implements a similar process model to both Chrome and OP and can also isolate content within a single page, which we discuss in Section 7.4. Gazelle calls

their process model *process-per-principal-instance* [Wang et al. 2009] and a web page instance is composed of two processes, one for plugins and one for JavaScript and HTML. Similar to OP and Chrome, navigation to new pages and tabs are contained by a web page instance; however, navigation between pages at the same origin does not create a new web page instance. Wang et al. [2009] do not specifically address new windows created by JavaScript, though based off the browser design, same-origin windows would be run in the same web page instance, similar to Chrome.

In OP2, we use a web page instance per page similar to all three browsers; however, we have chosen to combine each of the components in a web page instance into a single process with plugins remaining in a separate process. Navigation between pages causes new web page instances, and each tab is backed by a separate web page instance. New windows that are created by JavaScript are run in separate web page instances as in the original OP design. This change in architecture from OP to OP2 enables OP2 to reuse a substantial amount of WebKit [WebKit 2009] with minimal modifications to support the OP2 architecture. From a security perspective, the browser kernel still exposes the same set of system calls to each web page instance and is able to enforce modern browser security policy.

7.3 Displaying Page Content

OP uses VNC for rendering in one process and displaying the rendered content in another. This allowed OP to achieve isolation between the rendering engine and the user interface, a desirable property to prevent parsing and rendering bugs from interfering with the user's interaction with the browser. Gazelle uses a similar technique for displaying rendered content although does not use the VNC protocol [Wang et al. 2009].

Chrome implements a different technique for rendering in one process and displaying in another, using the rendering process to render the content, and displays the bitmap in a different process.

OP2 eliminates the VNC process for display to combine as many of the processes in a web page instance as we can. In place of VNC based rendering, OP2 uses native window reparenting supported by the window manager (i.e., Xorg, X11, or Windows). Initially, we designed the rendering and display engine similar to Chrome, using the Qt toolkit [Qt Software 2009] and encountered severe performance problems with our implementation. Rather than redesign and implement the rendering system used by Chrome, we chose to use window reparenting techniques natively supported by the window manager. Window reparenting is fast and requires no redirection of rendered content between processes, so it is as fast as displaying windows natively. It is also simple and directly supported by the Qt toolkit. The primary disadvantage is that the web page instance requires direct interaction with the window manager, something that Chrome is able to avoid. Direct interaction with the window manager limits OP2 to specific operating systems and exposes the window manager to potentially compromised browser components.

7.4 Cross-Origin IFRAME and Display Security

Display security refers to the browser's ability to isolate different-origin content inside of a single page. In all three browsers – OP, Chrome and Gazelle – the browser is able to isolate plugins in a separate process and quarantine vulnerabilities and bugs in plugins. Gazelle introduced frame isolation that provides isolation for cross-origin frames in web pages [Wang et al. 2009]. Gazelle's mechanism works by creating a new web page instance for the frame if it belongs to a different origin than that of the including page. This is a contrast from OP and Chrome where all content for a

single page is run using a single web page instance without differentiating based on the origin of the content. Gazelle was unable to show that this can be done efficiently, adding 2.6 seconds to the loading time of nytimes.com, and we presume the added delay is caused by the indirection of rendering and IE interposition layers [Wang et al. 2009].

In OP2, we adopted the Gazelle design and isolate different-origin frames and plugins using separate web page instances. In OP2, if a frame is encountered by the web page instance, a new web page instance is automatically created. Then, using window reparenting techniques, the frames are placed in the correct position on the page. Like Gazelle, OP2 isolates frames and uses content sniffing [Barth et al. 2009] to prevent a compromised web page instance from reading cross-origin HTML content. We cannot prevent a compromised web page instance from requesting sensitive cross-origin scripts, CSS, or images, since this is allowed within same-origin policy and available to a uncompromised web page instance. Gazelle has similar limitations, with the added ability to render cross-origin images in separate web page instances.

With some small modifications, OP2 implements the "2-D display delegation policy" described by Gazelle. The display policy prohibits the parent page from displaying any content over a cross-domain frame or plugin. This display policy is enforced in OP2 by the browser kernel. We show in Section 8 that this configuration is not only practical, but improves performance in OP2 for nytimes.com. This policy does differ from popular browsers, such as Firefox and Internet Explorer, and since this policy is more restrictive it can cause incompatibilities in the display of a page.

7.5 Increasing Compatibility

OP2 has increased compatibility for web pages compared to OP. While some benefit comes from our use of the WebKit HTML and JavaScript engines, we have devoted significant effort to make OP2 compatible with many different web pages. In order to do this we have included additional support for JavaScript to use our messaging API. For example, the XMLHttpRequest object is able to use the OP2 network component through the messaging API. We have also provided access to cookies within our storage component from JavaScript again by using the messaging API.

Plugins have also received significant compatibility work. Currently we support any plugin that uses the Netscape Plugin API (e.g., Flash, Java runtime, and Silverlight). Our previous implementation required customized support for each plugin to redirect calls into our messaging infrastructure. We recognize there are potential compatibility problems with plugin policies that place restrictions on plugins such as Flash and are currently working on a more permissive policy. Loosening the plugin security policy can open the door for potential exploit, and we are currently working on developing new browser policies that provide greater compatibility for plugins and evaluating the cost to compatibility [Grier et al. 2009].

7.6 Optimizations in OP2

The original version of OP performed approximately as fast as Firefox 2 [Grier et al. 2008]; however, since then browsers have seen substantial enhancements that improve performance. In our preliminary implementation of OP2, we found that OP2 added measurable overhead when compared to other WebKit-based browsers. The problem was that WebKit improved performance so significantly that the latency we added to certain key operations, such as downloading and displaying content for a new web page, caused a noticeable delay in the overall page load latency time.

The fundamental issues in the original version of OP2 that caused increased overhead were using new processes for each individual web page instance and serializing

slow operations, such as windows reparenting. By using new processes for each individual web page we had to pay the process initialization cost each time a user visited a new web site, and we precluded ourselves from using any WebKit optimizations that assumed process reuse, such as in-memory object caches. By serializing window reparenting, we added overhead directly to page load latency times.

To reduce this overhead, we optimized our web page instance management and parallelized slow serial operations. Specifically, we improved the load time of web pages in OP2 by using process pre-creation, parallelizing window manager operations, caching previously-used processes, and loading frames in parallel. The key to our optimizations is that we improve performance without compromising the security assurances of our OP2 architecture.

7.6.1 Process Pre-Creation. OP2 creates a new web page instance for each page and each web page instance requires a new process. During normal use, multi-process browser architectures such as OP, Chrome, and Gazelle all use more processes than their monolithic ancestors. As a result, the time it takes to initialize a new process can add overhead. To avoid this added overhead, we pre-create web page instances during idle times and use these pre-created web page instances to service new web page requests, thus avoiding process initialization overhead for new web page instances.

7.6.2 Parallelizing Window Manger Operations. As discussed in Section 7.3, OP2 uses window manager reparenting to combine the display from multiple processes. During our testing of OP2 we found that this design decision requires around 0.51 seconds to complete. In order to eliminate this time from the overall page load time, we perform the window reparenting asynchronously and allow the web page instance to load while the window manager reparents the window. Although this optimization does not eliminate the cost of reparenting windows, it does mask the cost while the browser loads the page.

7.6.3 Process Caching. WebKit uses an in-memory cache to provide fast retrieval of objects, such as scripts and stylesheets. This object cache improves the page load latency times by reusing web page resources shared between pages on subsequent visits. Since OP2 creates new processes for each web page, our original design was unable to use the object cache that WebKit provides.

To take advantage of the in-memory object cache, we implemented a process cache that reuses old web page instances for new web pages. When the browser navigates away from a web page, the previous web page instance could be killed and cleaned up by the browser. However, rather than removing these old web page instances, we add them to a cache that the browser kernel uses to service new web page instances. Each time the browser visits a new page, the browser kernel will first check the process cache to see if there are any web page instances that can be reused. If the browser kernel finds a suitable web page instance, then it is used for the new request, thus enabling the use of the in-memory object cache. If no suitable web page instances are found, then it will use a web page instance from the pre-created process pool for the new request.

One design decision we had to make was determining how to calculate cache hits for our process cache. One way we could have calculated cache hits was by matching the origin of the request and the origin of cached web page instances. For example, if there was a web page instance in the process cache that was used for docs.google.com and the browser issued a request for gmail.google.com, then it would reuse the docs.google.com web page instance since both are from google.com. This approach has the advantage of improving the hit rate by matching pages liberally, but has the disadvantage of

carrying around the state from the previous page, potentially leaking information to the new web page instance. Instead, our approach is to calculate cache hits based on the full URL of the request. This approach has the advantage of carrying around state for only the exact web page that was requested, thus minimizing potential information leakage, but will have a lower hit rate than an origin-based scheme.

7.6.4 Parallelizing Frames. Section 7.4 describes how OP2 performs frame isolation and display protection. In addition to improving security, isolating frames and running each frame in a separate web page instance can also improve performance. Since frames from different origins run in separate processes, OP2 naturally parallelizes the execution and rendering of frames. This additional parallelism can mask the latency added by our frame isolation techniques and can help improve performance on today's multicore systems.

8. EVALUATION

In this section, we evaluate the performance of OP2, and we present a qualitative security analysis of our browser. We have previously evaluated the performance of the original version of OP [Grier et al. 2008] and found that OP2 performed as well as Firefox 2 when measuring page load times. All measurements presented in this section were done with the latest version of OP2 as discussed in Section 7.

8.1 Performance Evaluation

To evaluate the OP2 web browser performance, we measure page load times and memory footprint. To determine the file system impact from the extensive logging, we examine the size of the audit logs for single-page loads. Our goal when performing these evaluations is only to verify that the browser does not introduce unreasonable delays that are noticeable by the user. We also aim to check that the logs for a running browser are reasonable in size. All experiments were carried out on a 2.66 GHz Intel Core 2 Duo with 8 GB of memory and a 250 GB serial ATA hard drive. The OS is Fedora Core 10, running the 64-bit version of the Linux kernel 2.6.27. OP2 uses version 4.5 of the Qt toolkit for the UI as well as other browser processes.

To measure the latency introduced by OP2, we compare the load times of a few common pages with those of Arora [2009]. Arora is a WebKit-based browser with a single-process browser architecture. Figure 14 shows a list of the sites tested and the loading latency times. All experiments have a standard deviation of less than 4% with a 0.95 confidence interval. Each page is loaded ten times, and the loading times are averaged. We monitor similar events inside of the OP2 browser's web page subsystem and in the event system in Arora. In all tests, we use a warmed up browser that has previously loaded the page to warm any caches.

8.1.1 Optimizations. Figure 14 shows the results of five different OP2 configurations in comparison to Arora. We have performed three optimizations to improve the performance of OP2 and evaluate each optimization as well as the effects of frame isolation in this section. Figure 14 shows how each optimization improves the overall OP2 performance.

Process Pre-Creation. Our first optimization is designed to eliminate the cost of creating new processes by creating web page instances during idle times in anticipation of future use. Currently, we keep a web page instance pool containing a single web page instance. As shown by the second bar from the left in Figure 14, OP2 with only the process pre-creation optimization performs slower than Arora in every test.

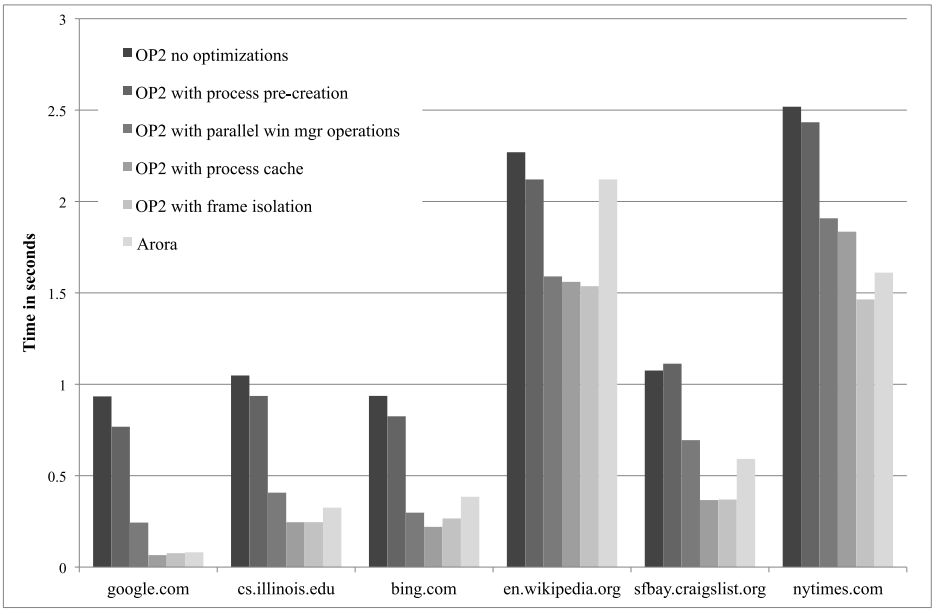

Fig. 14. Loading latencies for a six popular pages using OP2 and Arora. From the left bar to the right shows OP2 starting without any optimizations, and shows the effects of adding each optimization cumulatively, where the OP2 with frame isolation bar represents a fully-optimized OP2 browser.

Parallelizing Window Manger Operations. The second optimization we implemented performs the window reparenting (Section 7) in parallel with the page downloading and rendering. This optimization provides the largest performance improvement. The third bar from the left in Figure 14 shows the impact of this optimization combined with process pre-creation. On average, parallelizing the window manager operations improves page load times by 0.51 seconds and the largest improvement is 0.53 seconds for *en.wikipedia.org*.

Process Caching. Our last optimization reuses old web page instances if the same URL is loaded multiple times, limiting the costs of initializing a web page instance with the same content. Process caching uses a least recently used cache of web page instances that can be reused when a previously seen URL is loaded. The default cache configuration retains ten web page instances in the cache.

Combined with the other two optimizations, process caching improves the performance of OP2 to be as fast or faster than Arora in all of our tests, as shown by the fourth bar from the left in Figure 14. Process pre-creation improving the page load time by 0.14 seconds on average with the largest improvement of 0.33 seconds for cs.illinois.edu.

Frame Isolation. In Section 7, we discussed the design of OP2 that allows for the browser to automatically isolate frames. The fourth and fifth bars from the left in Figure 14 show a comparison of OP2 without and with frame isolation. For most of the pages tested, we see little difference in the page load times; however, for nytimes.com the load time with frame isolation is noticeably faster than without frame isolation. This performance gain results because nytimes.com uses different-origin frames. OP2 isolates different-origin frames in separate web page instances allowing

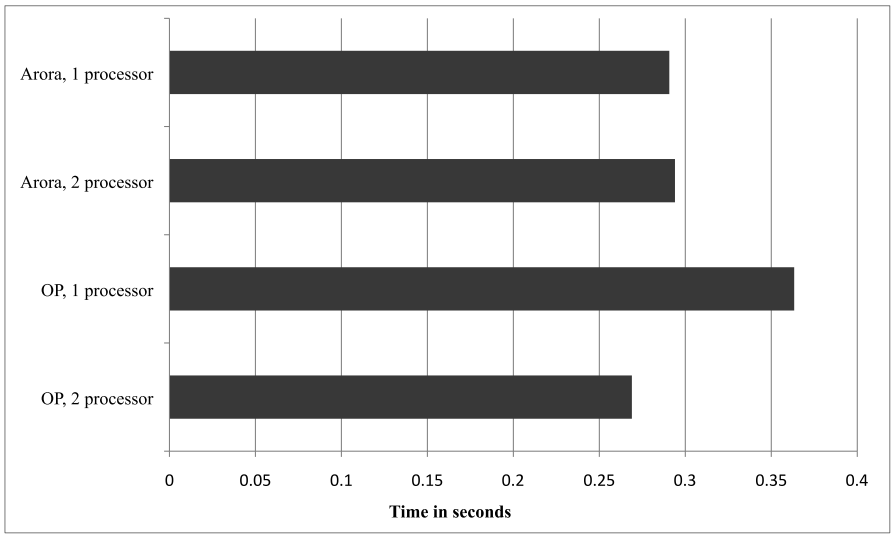

Fig. 15. Loading latencies cs.illinois.edu in Arora and OP2 on a single and dual processor configuration.

OP2 to use additional CPUs in our system to render frames in parallel, thus improving performance further.

Effect of Optimizations. As shown in Figure 14, the versions of OP2 without optimizations are always slower than Arora, sometimes by as much as 11x; however, with all optimizations enabled, OP2 is as fast and sometimes faster than Arora. OP with the first three optimizations is up to 0.87 seconds faster than OP without optimizations. With frame isolation, OP2 is able to render nytimes.com 0.14 seconds faster than Arora and 1.05 seconds faster than OP2 without any optimizations. The results in Figure 14 indicate that with our optimizations OP2 does not introduce latency that would be detrimental to a user using OP2.

Parallelism in OP2. To determine the impact of parallelism on the performance of OP2, we have tested OP2 and Arora with a single processor and with two processors. Figure 15 shows the results of testing the two different configurations while loading cs.illinois.edu. In our tests, OP2 loads our test page 94 ms faster with two processors than one while Arora performs approximately equal for both configurations, demonstrating that OP2 is able to use additional processors to improve browser performance.

By decomposing the browser into separate subsystems that run in different OS processes, we add latency to individual operations by requiring IPC calls. For example, sending an HTTP request in Arora and OP2 both require setting up HTTP headers, attaching appropriate cookies, connecting to the network, and sending the request. However, in OP2, the web page instance must also send an IPC message, through the browser kernel, to the network subsystem, adding latency directly to roughly equivalent functionality.

The added latency of individual operations exposes parallelism that is absent in current versions of WebKit, potentially enabling multicore systems to improve performance. For example, OP2 will overlap the handling of HTTP requests with the downloading, parsing, rendering, and painting of WebKit. We believe that this trade off of increased latency for increased parallelism is why OP2 is able to run faster than Arora on our multicore system. We recognize that this observation might indicate that

Table III. Memory Footprint per Process for OP2 Loading a Single Web Page

Process	Memory size	Purpose
op2	12.5 MB	current web page instance
op2	6 MB	pre-created web page instance
log_storage	496 KB	audit log
op-ui	7.9 MB	user interface
cookie	4.7 MB	cookie storage
network	5.9 MB	network and cache
kernel	272 KB	browser kernel
Total	38 MB	

Table IV. Audit Log Size Generate for a Single Visit

Site	Audit log size (KB)
bing.com	26
google.com	23
craigslist.org	20
cs.illinois.edu	175
wikipedia.org	156

WebKit could be made faster by introducing more parallelism, which could expose the latency added by OP2, but further study of this performance issue is beyond the scope of this paper.

8.2 Memory Usage

To measure the memory footprint of the browser, we load the browser and navigate to a single web page with no plugins. We use the Gnome system monitor to measure memory usage. The results are displayed in Table III. The single-page snapshot of OP2 contains three processes used by every web page instance: the cache and network, user interface, cookie, and audit log components; the web page instance consists of a single process responsible for the rendering the page content. The total memory used by OP2 for a single web page visit is 38 MB. Each web page instance requires at least an additional 6 MB of memory and the memory used by the web page instance will vary depending on the complexity of the page being rendered. This number does not include individual plugin instances, since pages will vary greatly due to their use of third-party plugins.

8.3 Log Size

We also examine the footprint of the audit log that is generated for each site. The audit log is stored as a single file consisting of all the events recorded during a page load inside of OP2. After each test, the audit log is cleared and the browser restarted to provide a clean start. As can be seen in Table IV, the amount of space needed to record events is small for a single page. The size is largely dependent on the size of the pages being downloaded rather than data introduced for audit purposes. To quantify the likely storage needed for typical browsing, one of the members of the research team used OP2 for typical browsing needs over several weeks. Over this period of time the user created over 1,000 web page instances and accumulated an audit log of 206 MB, which is reasonable considering the low cost of storage.

8.4 Security Analysis

In addition to performance, we also evaluate the OP2 browser's resilience to attacks. Isolation in the OP2 browser makes many attacks considerably more difficult and lessens the impact of exploits.

8.4.1 Memory Attacks. Code execution is one class of vulnerabilities common in browsers and plugins. The OP2 browser reduces the severity of this class of attack significantly. Our SELinux policies limit a compromised OP2 subcomponent so that it can send messages only to other subcomponents and perform limited interactions with the local system. Protecting the local operating system from an exploited browser is critical to provide security for the system and browser. Operating system sandboxing is provided today by most operating systems using a variety of techniques.

Our host sandboxing policies are designed to limit exploited browser components to sending messages through the browser kernel. If the attack results in malicious messages being sent inside the browser, the browser kernel enforces security policy and forces the exploit to comply with local security policy. Our use of formal methods provides additional assurance that an exploited process must adhere to browser security policy. Our security mechanisms do, however, allow exploited subcomponents to access information that security policy permits. For example, a compromised plugin can access cookies, DOM elements, and other browser resources available to correctly operating plugins.

8.4.2 Domain Isolation Bugs. In general, a web browser's ability to enforce separation between frames, pages, and windows is often subject to attack. Our browser kernel maintains origin labels and enforces access control policy on messages passed between processes. Since the message passing between processes is explicit, and since the browser kernel enforces our security policies directly, there is less surface for domain bugs to be exploited. In addition, our use of formal methods to verify the same-origin policy interactions between plugins and web page instances also asserts that domain bugs are not present in the model of our system.

8.4.3 Visual Spoofing. The design of OP2 also prohibits many forms of visual spoofing, which could result in phishing and other types of attacks. By isolating the browser chrome (i.e., UI widgets such as the status bar and the back button) from content rendering, we force the HTML parsing engine to send explicit messages to set the address and status bars. While we do not currently defend against malicious status bar modification, we do ensure that the page content domain is the same as the domain displayed in the address bar. In addition, the separation of browser chrome from content rendering reduces the chance of JavaScript or the HTML engine spoofing other visual components.

8.4.4 Web Application Bugs. OP2 does not provide protection for clients from web application bugs, which could result in attacks such as cross-site scripting or cross-site request forgery. OP2 can prevent the malicious script from making access to the local file system as well as preventing network access outside of the browser security policy. This can assist in combating the effects of some types of cross-site scripting attacks but does not prevent the vulnerable script from being exploited.

8.5 Compatibility Testing

Using OP2 we have performed rendering tests from the Acid test suites [Hickson 2009]. Figure 16 shows screen shots of each of the Acid test results. These tests primarily exercise the JavaScript, HTML, and CSS rendering in browsers, particularly edge cases

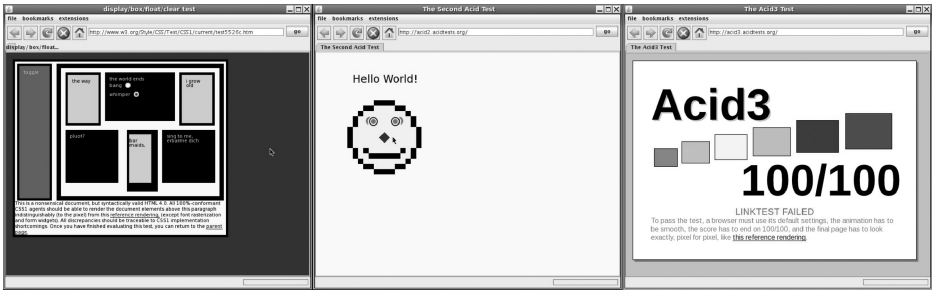

Fig. 16. Visual test results from Acid 1, 2, and 3 tests. Each of the images is a screen shot of the OP2 browser rendering the results of the Acid test.

that demonstrate a browser's ability to render the majority of pages on the web. In OP2, the Acid tests primarily check functionality provided by WebKit. The third Acid test scores a 100/100, the same as for the version of WebKit we use. For comparison, Firefox 3 scores 71, Firefox 2 scores 53, and Google Chrome (on Windows Vista) scores 79. The results of these tests show that the architecture and implementation of OP2 does not introduce incompatibilities with today's web standards.

As stated in Section 7.4, OP2 implements a more restrictive display policy than popular browsers, such as Firefox and Internet Explorer, and this policy can cause incompatibilities in the rendering of web pages. In Wang et al. [2009], the 2-D display delegation policy causes incompatibilities with six of the top 100 sites as ranked by Alexa.

9. RELATED WORK

Our OP web browser introduces a new architecture for building more secure web browsers. The most closely related works are Tahoma [Cox et al. 2006], the Building a Secure Web Browser project [Ioannidis and Bellovin 2001], Google Chrome, Gazelle [Wang et al. 2009], and a paper by Reis and Gribble [2009], all of which propose and analyze new browser architectures. Tahoma shares many of the same design principles as OP, but our architecture differs in two key ways. First, Tahoma uses VMMs to provide isolation for different web-based applications, and a manifest to help craft their network policy. In contrast, we use OS-level mechanisms for isolation, and we support existing web-based applications. OP tracks interactions at a finer level of granularity, allowing us to explore novel policies such as our plugin policy. However, these two architectures are complementary: one could imagine OP using Tahoma to provide even stronger isolation. In the Building a Secure Web Browser project [Ioannidis and Bellovin 2001], the authors propose a new browser architecture that relies on the underlying OS policies (e.g., file system permissions) to enforce browser security. OP handles security policies in the browser kernel, giving us more flexibility.

Google Chrome is a recently released browser that uses operating system processes to separate instances of page rendering in different ways. Chrome provides four different process creation models that can provide isolation between different browser entities [Google 2008]. Unlike OP2, Chrome does not support protections between frames in a web browser and does not enforce security policy for plugin content. Chrome also relies on the parsing and rendering engine to correctly control network requests, placing security decisions in the same process as the rendering engine. We separate the security enforcement from rendering and allow security decisions to be made by the browser kernel, allowing plugins to be subject to the policies enforced by the browser kernel.

A number of recent projects develop techniques for securing web-based applications [Chong et al. 2007b, 2007a; Erlingsson et al. 2007], securing JavaScript [Anupam and Mayer 1998; Jim et al. 2007; Reis et al. 2006; Yu et al. 2007], supporting mashups [Jackson and Wang 2007], protecting privacy [Jackson et al. 2006], enforcing the same-origin policy [Chen et al. 2007b; Karlof et al. 2007], adding new abstractions for improved sharing [Wang et al. 2007], overcoming DNS rebinding attacks in browsers [Jackson et al. 2007], and cross-frame communications [Barth et al. 2008]. These projects are orthogonal to secure web browser design; our goal is to provide a more secure platform to implement these, and other, techniques.

The idea of sandboxing browsers was first introduced by Goldberg et al. [1996], and GreenBorder is a commercial product that sandboxes Internet Explorer [GreenBorder 2007]. We use this type of sandboxing in our OP browser as the starting point for our security, and we focus on more fine-grained interactions within the browser itself. Plus, by breaking our browser into different components, we can apply different sandboxing rules to each subsystem, giving us even more control over our browser's interactions with the underlying system.

We show how our browser can be used to analyze browser-based attacks; current approaches for analyzing intrusions will not work for browser-based attacks. Several recent projects [Goel et al. 2005; King and Chen 2003; King et al. 2005] use OS-level dependency graphs to highlight the subset of activity on a system that is likely to be part of an attack. These techniques are effective for server-style workloads, where servers isolate distinct sessions into different OS-level processes, thus allowing them to track malicious sessions using OS-level states and events. However, these techniques fall short when there are long-lived processes that handle multiple sessions, because they taint entire OS-level objects conservatively and cannot separate out unrelated activities using OS-level events alone. In addition to OS-level techniques, researchers have been able to analyze browser-based attacks by using client-based honeypots [Moshchuk et al. 2006; Provos et al. 2007; Wang et al. 2006] to crawl the Internet looking for malicious sites. For example, the HoneyMonkey project [Wang et al. 2006] batches many different sites together, and when they detect a malicious site, they reprocess each site in isolation to determine which one was responsible for the attack. These techniques work well for automated crawling experiments but are not suitable for analyzing active browsers, since they fail to capture and integrate the interactions of the user – something OP handles well.

10. CONCLUSIONS

In this article, we have described the OP web browser and the different elements that make our browser secure. We have shown that by using an architecture that is designed to be secure we can enforce security policies that are flexible enough to apply to browser plugins, while at the same time formally verifying important security properties. We have also presented OP2, a refinement of the OP web browser architecture that incorporates new techniques developed since the development of OP.

The OP web browser is responsive to user interaction and implements features that make it compatible with current web pages. We include two plugins, supporting Flash-compatible content as well as other multimedia content, JavaScript, and basic web page support, giving us a functional browser capable of enforcing security policies. We have also included plugins into our security model, which are difficult for current browsers to control and enforce policy upon. Using the system's implementation, we have created and shown a formal model using Maude and model-checked invariants describing the security of our browser. We have also shown how the OP web browser can assist in forensic examination of attacks that we are unable to prevent.

All of these elements build on the security of the OP web browser, creating a web client capable of withstanding attack. We have demonstrated that, by design, the OP web browser is not vulnerable to many forms of browser attacks yet has the full functionality of the browser.

REFERENCES

ADOBE. Flash player settings manager. http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager.html.

ADOBE. Flash player update available to address security vulnerabilities. http://www.adobe.com/support/security/bulletins/apsb07-12.html.

ADOBE. 2008. External data not accessible outside a macromedia flash movie's domain. http://www.adobe.com/go/tn_14213.

ADOBE. 2009a. Adobe flash player. http://www.adobe.com/products/flashplayer/.

ADOBE. 2009b. Flash player penetration. http://www.adobe.com/products/player_census/flashplayer/.

ANUPAM, V. AND MAYER, A. 1998. Security of web browser scripting languages: Vulnerabilities, attacks, and remedies. In *Proceedings of the 7th USENIX Security Symposium*.

ARORA. 2009. Arora: Cross platform WebKit browser. http://code.google.com/p/arora/.

AUSCERT. Sun java runtime environment vulnerability allows remote compromise. http://www.auscert.org.au/render.html?it=7664.

BARTH, A., JACKSON, C., AND MITCHELL, J. C. 2008. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium*. 17–30.

BARTH, A., CABALLERO, J., AND SONG, D. 2009. Secure content sniffing for web browsers or how to stop papers from reviewing themselves. In *Proceedings of the IEEE Symposium on Security and Privacy*.

CHEN, S., MESEGUER, J., SASSE, R., WANG, H. J., AND WANG, Y.-M. 2007a. A systematic approach to uncover security flaws in GUI logic. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. 71–85.

CHEN, S., ROSS, D., AND WANG, Y.-M. 2007b. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. 2–11.

CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. 2007a. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*. 31–44.

CHONG, S., VIKRAM, K., AND MYERS, A. C. 2007b. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium*.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND QUESADA, J. F. 2002. Maude: Specification and programming in rewriting logic. *Theoret. Comput. Sci. 285*, 2, 187–243.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. 2007. Maude manual (version 2.3).

COX, R. S., HANSEN, J. G., GRIBBLE, S. D., AND LEVY, H. M. 2006. A safety-oriented platform for web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*. 350–364.

ERLINGSSON, U., LIVSHITS, B., AND XIE, Y. 2007. End-to-end web application security. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS XI)*.

GOEL, A., PO, K., FARHADI, K., LI, Z., AND DEL LARA, E. 2005. The Taser intrusion recovery system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 163–176.

GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. 1996. A secure environment for untrusted helper applications. In *Proceedings of the 1996 USENIX Security Symposium*. 1–13.

GOOGLE. 2008. Process models (Chromium developer documentation). http://dev.chromium.org/developers/design-documents/process-models.

GOOGLE. 2009. Google chrome. http://www.google.com/chrome.

GREENBORDER. 2007. Greenborder desktop DMZ solutions. http://www.greenborder.com.

GRIER, C., TANG, S., AND KING, S. T. 2008. Secure web browsing with the OP web browser. In *Proceedings of the IEEE Symposium on Security and Privacy*. 402–416.

GRIER, C., KING, S. T., AND WALLACH, D. S. 2009. How I learned to stop worrying and love plugins. In *Web 2.0 Security and Privacy*.

HICKSON, I. 2009. Acid tests - the web standards project. http://www.acidtests.org.

IOANNIDIS, S. AND BELLOVIN, S. M. 2001. Building a secure web browser. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*.

JACKSON, C. AND WANG, H. J. 2007. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the 16th International World Wide Web Conference (WWW)*. 611–620.

JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. C. 2006. Protecting browser state from web privacy attacks. In *Proceedings of the 15th International Conference on World Wide Web (WWW)*. 737–744.

JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. 2007. Protecting browsers from DNS rebinding attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*.

JIM, T., SWAMY, N., AND HICKS, M. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web*. 601–610.

JOVANOVIC, N., KIRDA, E., AND KRUEGEL, C. 2006. Preventing cross site request forgery attacks. In *Proceedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*.

KARLOF, C., TYGAR, J., WAGNER, D., AND SHANKAR, U. 2007. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. 58–71.

KDE. 2009. The konqueror web browser. http://www.konqueror.org/features/browser.php.

KING, S. T. AND CHEN, P. M. 2003. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 223–236.

KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. 2005. Enriching intrusion alerts through multi-host causality. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

LAMPSON, B. W. 1973. A note on the confinement problem. *Comm. ACM 16*, 10, (Oct.), 613–615.

LOSCOCCO, P. AND SMALLEY, S. 2001. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the USENIX Annual Technical Conference FREENIX Track*.

MESEGUER, J. 1992. Conditional rewriting logic as a united model of concurrency. *Theoret. Comput. Sci. 96*, 73–155.

MICROSOFT. Activex security: Improvements and best practices. http://msdn2.microsoft.com/en-us/library/bb250471.aspx.

MOSHCHUK, A., BRAGIN, T., GRIBBLE, S. D., AND LEVY, H. M. 2006. A crawler-based study of spyware on the web. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

MOZILLA. 2004. Netscape plugin API. http://www.mozilla.org/projects/plugins/.

MOZILLA. 2009. Rhino: Javascript for java. http://www.mozilla.org/rhino/.

NOVELL. 2009. Apparmor Linux application security.

PETRKOV, P. D. Pdf pwns windows. http://www.gnucitizen.org/blog/0day-pdf-pwns-windows.

PETRKOV, P. D. Quicktime pwns firefox. http://www.gnucitizen.org/blog/0day-quicktime-pwns-firefox.

PROVOS, N. 2003. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium*. 257–272.

PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. 2007. The ghost in the browser: Analysis of Web-based malware. In *Proceedings of the Workshop on Hot Topics in Understanding Botnets (HotBots)*.

PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE, F. 2008. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*. 1–15.

QT SOFTWARE. 2009. Qt – a cross-platform application and UI framework. http://www.qtsoftware.com.

REIS, C. AND GRIBBLE, S. D. 2009. Isolating web programs in modern browser architectures. In *Proceedings of the EuroSys conference*.

REIS, C., DUNAGAN, J., WANG, H., DUBROVSKY, O., AND ESMEIR, S. 2006. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*.

RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. 1998. Virtual network computing. *IEEE Internet Comput. 2,* 1 (Jan.), 33–38.

SOPCAST. 2009. Sopcast. http://www.sopcast.org/.

STAMOS, A. AND LACKEY, Z. 2006. Attaking ajax web applications. *Presented at the Black Hat USA Conference.*

SUN. Java security architecture.
http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc1.html.

TURNER, D. 2007. Symantec internet security threat report: Trends for january - june 07.
http://www.symantec.com/business/theme.jsp?themeid=threatreport.

WANG, H. J., FAN, X., HOWELL, J., AND JACKSON, C. 2007. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*.

WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. 2009. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the USENIX Security Symposium*.

WANG, Y.-M., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. 2006. Automated web patrol with strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*.

WEBKIT. 2009. The webkit open source project.http://www.webkit.org.

YU, D., CHANDER, A., ISLAM, N., AND SERIKOV, I. 2007. JavaScript instrumentation for browser security. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 237–249.

Received September 2008; revised August 2009, January 2010, August 2010; accepted September 2010