Isolating Web Programs in Modern Browser Architectures

Charles Reis Steven D. Gribble
University of Washington / Google, Inc. *
{creis, gribble}@cs.washington.edu

Abstract

Many of today's web sites contain substantial amounts of client-side code, and consequently, they act more like programs than simple documents. This creates robustness and performance challenges for web browsers. To give users a robust and responsive platform, the browser must identify program boundaries and provide isolation between them.

We provide three contributions in this paper. First, we present abstractions of web programs and program instances, and we show that these abstractions clarify how browser components interact and how appropriate program boundaries can be identified. Second, we identify backwards compatibility tradeoffs that constrain how web content can be divided into programs without disrupting existing web sites. Third, we present a multi-process browser architecture that isolates these web program instances from each other, improving fault tolerance, resource management, and performance. We discuss how this architecture is implemented in Google Chrome, and we provide a quantitative performance evaluation examining its benefits and costs.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures; D.4.5 [*Operating Systems*]: Reliability—Fault tolerance; H.4.3 [*Information Systems Applications*]: Communications Applications—Information browsers

General Terms Design, Experimentation, Performance, Reliability

Keywords Web browser architecture, isolation, multi-process browser, reliability, robustness

* This work was partially performed at Google while the first author was a graduate student intern and the second was on sabbatical.

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.

1. Introduction

Today's publishers are deploying web pages that act more like programs than simple documents, and these programs are growing in complexity and demand for resources. Current web browser architectures, on the other hand, are still designed primarily for rendering basic pages, in that they do not provide sufficient isolation between concurrently executing programs. As a result, competing programs within the browser encounter many types of interference that affect their fault tolerance, memory management, and performance.

These reliability problems are familiar from early PC operating systems. OSes like MS-DOS and MacOS only supported a single address space, allowing programs to interfere with each other. Modern operating systems isolate programs from each other with processes to prevent these problems.

Surprisingly, web browsers do not yet have a program abstraction that can be easily isolated. Neither pages nor origins are appropriate isolation boundaries, because some groups of pages, even those from different origins, can interact with each other within the browser. To prevent interference problems in the browser, we face three key challenges: *(1)* finding a way for browsers to identify program boundaries, *(2)* addressing the complications that arise when trying to preserve compatibility with existing web content, and *(3)* rearchitecting the browser to isolate separate programs.

In this paper, we show that web content can in fact be divided into separate web programs, and we show that separate instances of these programs can exist within the browser. In particular, we consider the relationships between web objects and the browser components that interact with them, and we define web program instances based on the access control rules and communication channels between pages in the browser. Our goal is to use these abstractions to improve the browser's robustness and performance by isolating web program instances. We find they are also useful for reasoning about the browser's execution and trust models, though we leave security enhancements as a goal for future work.

We show that these divisions between web program instances can be made without losing compatibility with existing content or requiring guidance from the user, although doing so requires compromises. We define a web program as pages from a given *site* (i.e., a collection of origins sharing

the same domain name and protocol), and a web program instance as a *site instance* (i.e., pages from a given site that share a communication channel in the browser). Compatibility with existing web content limits how strongly site instances can be isolated, but we find that isolating them can still effectively address many interference problems.

To better prevent interference between web program instances, we present a browser architecture that uses OS processes as an isolation mechanism. The architecture dedicates one process to each program instance and the browser components required to support it, while the remaining browser components run safely in a separate process. These web program processes leverage support from the underlying OS to reduce the impact of failures, isolate memory management, and improve performance. As a result, the browser becomes a more robust platform for running active code from the web. Web program processes can also be sandboxed to help enforce some aspects of the browser's trust model, as we discuss in a related technical report [Barth 2008].

Google has implemented the architecture described above in the open source Chromium web browser. The Google Chrome browser is based on the Chromium source code; we will refer to both browsers as Chromium in this paper. While at Google, the first author of this paper helped add support for site instance isolation to Chromium's multi-process architecture, allowing each site instance to run in a separate process. While the current implementation does not always provide strict isolation of pages from different sites, we argue that it achieves most of the potential benefits and that strict isolation is feasible.

We evaluate the improvements this multi-process architecture provides for Chromium's robustness and performance. We find that it provides a more robust and responsive platform for web programs than monolithic browsers, with acceptable overhead and compatibility with existing web content.

The rest of this paper is organized as follows. We present ideal program abstractions and show how they can be applied to real web content and browser components in Section 2, along with how these abstractions are limited by backwards compatibility. In Section 3, we present a multi-process architecture that can isolate these abstractions. We also describe Chromium's implementation of the architecture and how it can address concrete robustness problems. We evaluate the benefits and costs of the architecture in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2. Finding Programs in Browsers

Many robustness problems could be solved if independent web-based programs were effectively isolated by the browser's architecture. Crashes and leaks could be contained, and different programs in the browser could accomplish work concurrently. This isolation requires strong boundaries between programs, but we currently lack a precise program abstraction for browsers. As a result, most browsers have monolithic architectures in which all content and browser components are combined in one address space and process (see Figure 1).

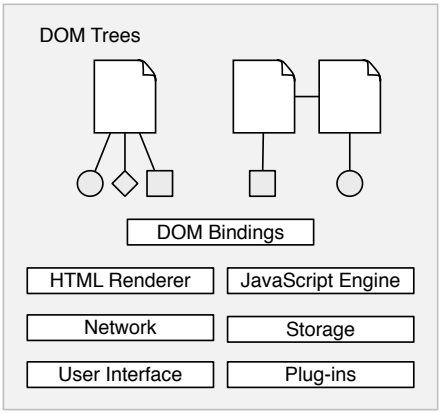


Figure 1. All web content and browser components share fate in a single process in monolithic browser architectures, in contrast to the multi-process architecture in Figure 5.

Unfortunately, it is challenging to find an appropriate way to define program boundaries in today's browsers. Consider a straw man approach that treats each web page as a separate program, isolating pages from each other. This approach breaks many real web programs that have multiple communicating pages. For example, mapping sites often allow a parent page to script a separate map page displayed in a frame. Similarly, calendar pop-up windows are frequently used to fill in dates in web forms. Isolating every page would break these interactions.

Origins are another inadequate straw man. Two copies of the same page are often independent and can be isolated from each other, while two pages from partially differing origins can sometimes access each other. Thus, isolating pages based solely on origin would be too coarse in some situations and too fine grained in others.

In this section, we show that it is possible for a browser to identify program boundaries in a way that lets it isolate program instances from each other, while preserving backwards compatibility. We provide ideal abstractions to capture the intuition of these boundaries, and we provide concrete definitions that show how programs can be defined with today's content, as summarized in Figure 2. We then discuss the complications that arise due to the browser's execution and trust models, and what compromises are required to maintain compatibility with existing content.

2.1 Ideal Abstractions

We first define two idealized abstractions that represent isolated groups of web objects in the browser.

Web Program	A set of related web pages and their sub-resources that provide a common service.
Web Program Instance	Copies of pages from a web program that are tightly coupled within the browser.
Site	A concrete way to define a web program based on access control between pages.
Browsing Instance	A set of connected windows and frames in the browser that share DOM-based communication channels.
Site Instance	A concrete way to define a web program instance, using the set of same-site pages within a browsing instance.

Figure 2. Ideal abstractions (bold) and concrete definitions (italic) to identify program instances within browsers.

A *web program* is a set of conceptually related pages and their sub-resources, as organized by a web publisher. For example, the Gmail web program consists of a parent page, script libraries and images, pages in embedded frames, and optional pop-out windows for chatting and composing messages. The browser combines all of these web objects to form a single coherent program, and it keeps them distinct from the objects in other web programs.

Web programs are easy to understand intuitively but difficult to define precisely. Some research proposals have called for explicit mechanisms that enumerate the components of each web program [Reis 2007b, Cox 2006], but in this paper we seek to infer web program boundaries from existing content without help from web publishers or users.

Browsers make it possible to visit multiple copies of a web program at the same time. For example, a user may open Gmail in two separate browser windows. In general, these two copies do not communicate with each other in the browser, making them mostly independent. We introduce a second abstraction to capture this: a *web program instance* is a set of pages from a web program that are connected in the browser such that they can manipulate each other's contents. Pages from separate web program instances cannot modify each other's contents directly and can be safely isolated.

2.2 Concrete Definitions

The browser can only isolate instances of web programs if it is able to recognize the boundaries between them. To achieve this, we show how to concretely define web programs using *sites*, and how to define web program instances using *site instances*.

Sites The browser already distinguishes between unrelated pages with its access control rules, using the Same Origin Policy [Ruderman 2001]. This policy permits some pages to communicate within the browser, so any practical program boundaries must take it into account. To reflect this, we define web programs based on which pages may be able to access each other.

Taking this approach, pages permitted to interact should be grouped together into web programs, while pages that are not should be isolated from each other. We focus on pages and not other types of web objects because the Same Origin Policy bases its access control decisions on the origin (i.e., protocol, full host name, and port) of each page. Sub-resources in a page, such as script libraries, are governed based on the origin of their enclosing page, not their own origin. Note that pages loaded in frames are considered separate from their parent pages, both by the Same Origin Policy and by our definitions.

When pages are allowed to interact, they do so by accessing each other's Document Object Model (DOM) trees. DOM trees provide a representation of each page's contents that can be manipulated by script code. Within the browser, particular components are responsible for supporting this behavior and enforcing access control. The HTML rendering component generates DOM trees from pages. Script code runs within the JavaScript engine and can only interact with these trees via the DOM bindings component, which acts as a reference monitor. The bindings allow a page's code to manipulate and call functions in other pages from the same origin. If the origins do not match, the code is denied access to the page's DOM tree and can only use a small API for managing the page's window or frames. This means that pages from different origins can generally be isolated from each other.

However, origins do not provide perfect program boundaries because a page can change its origin at runtime. That is, a page's code can modify its `document.domain` property, which the DOM bindings use for access control checks. Fortunately, this property can only be modified within a limited range: from sub-domains to more general domains (e.g., from `a.b.c.com` to `c.com`) and only up to the "registry-controlled domain name." A registry-controlled domain name is the most general part of the host name before the public suffix (e.g., `.com` or `.co.uk`) [Mozilla 2007]. Note that any port specified in a URL (e.g., 8080 in `http://c.com:8080`) becomes irrelevant for access control checks when a page changes its origin, so pages served from different ports may also access each other.

Since a page's origin can change, we instead define a web program based on the range of origins to which its pages may legally belong. We denote this as a *site*: the combination of a protocol and registry-controlled domain name. Sites provide a concrete realization of the web program abstraction. Pages from the same site may need the ability to access and modify each other within the browser, while pages from different sites can be safely isolated from each other.

While effective and backwards compatible, using sites for isolation does represent a somewhat coarse granularity, since this may group together logically independent web programs

hosted on the same domain. We discuss the implications of this further in Section 2.4.

Browsing Instances Defining web program instances requires knowing which pages share communication channels inside the browser. A page can only access the contents of other pages if it has references to them, and the browser's DOM bindings only provide such references in certain cases. Thus, we must consider how groups of communicating pages are formed.

DOM-based communication channels actually arise between the containers of pages: windows[1] and frames. We say two such containers are *connected* if the DOM bindings expose references to each other. This occurs if a page opens a second window, where the first page is returned a reference to the second, and the second can access the first via `window.opener`. This also occurs if a child frame is embedded in a parent page, where they become accessible via `window.frames` and `window.parent`. Connections match the lifetime of the container and not the page: if a window or frame is navigated to a different page, it will still have a reference to its parent or opener window.

These persistent connections imply that we can divide the browser's page containers into connected subsets. We define a set of connected windows and frames as a *browsing instance*, which matches the notion of a "unit of related browsing contexts" in the HTML 5 specification [Hickson 2008]. New browsing instances are created each time the user opens a fresh browser window, and they grow each time an existing window creates a new connected window or frame, such as a chat window in Gmail. Because browsing instances are specific to containers and not pages, they may contain pages from multiple web programs (i.e., sites).

In current browsers, named windows are another way to obtain a reference to a page's container. A page can attempt to open a new window with a given name, and if the browser finds an existing window with the same name, it can return a reference to it instead of creating a new window. In theory, this allows any two windows in the browser to become connected, making it difficult to isolate browsing instances. However, this feature is not covered in the HTML 4 specification, and its behavior is left to the user agent's discretion in HTML 5. Specifically, browsers are permitted to determine a reasonable scope for window names [Hickson 2008]. To allow web program instances to be isolated, we recommend shifting from a global namespace for windows to a separate namespace for each browsing instance. This change will have little impact on most browsing behavior, and it more intuitively separates unrelated windows in the browser.

Pages do share some other communication channels in the browser, but not for manipulating DOM trees. For example, pages from the same origin may access a common set of cookies using the browser's storage component. Such channels can continue to work even if browsing instances are isolated from each other.

[1] We consider windows and tabs equivalent for this purpose.

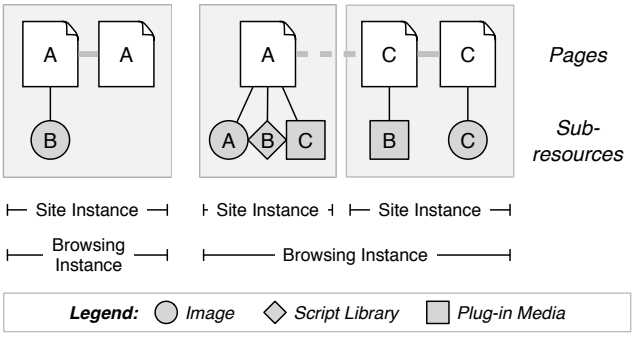


Figure 3. We divide web content into independent site instances, based on the sites of pages (e.g., A, B, C) and the connections between page containers (horizontal gray lines).

Site Instances We can further subdivide a browsing instance into groups of pages from the same web program, for a concrete realization of a web program instance. Specifically, we define a *site instance* as a set of connected, same-site pages within a browsing instance. Since all pages within a browsing instance are connected, there can be only one site instance per site within a given browsing instance.

We show an example of how web content in the browser can be divided into browsing instances and site instances in Figure 3. Each site instance contains pages from a single site but can have sub-resources from any site. All pages in a browsing instance can reference each other's containers, but only those from the same site share DOM access, as shown by the solid and dashed gray lines between pages.

Note that the same site can have multiple site instances in separate browsing instances (e.g., site A). In this case, pages from different site instances have no references to each other. This means that they have no means to communicate, even though the Same Origin Policy would allow them to.

Similarly, a single browsing instance may contain site instances from different sites (e.g., sites A and C). In this case, pages in different site instances do have references to each other, but the Same Origin Policy does not allow them to access each other's DOM trees.

In both cases, pages in separate site instances are independent of each other. This means that site instances can be isolated from each other in the browser's architecture without disrupting existing web content.

It is also worth noting that the browsing instance and site instance boundaries are orthogonal to the groupings of windows and tabs in the browser. It is unfortunately not possible to visually identify a browsing instance by looking at a set of browser windows, because a single window may contain tabs from different browsing instances. This is because a single browsing instance may have connected tabs in sep-

arate windows. This is especially true in browsers that allow tabs to be dragged to different windows, such as Chromium. As a result, the browser's user interface does not reflect the boundaries between web program instances.

Summary With these concrete definitions, we have identified precise boundaries between groups of web objects in the browser. We use sites as a proxy for web programs, based on whether documents can access each other. We use site instances as a proxy for web program instances, based on whether those documents' containers (e.g., windows) can access each other. Each of these definitions can act as an architectural boundary in the browser, with various strengths that we discuss in Section 3.2.

2.3 Practical Complications

Some aspects of the browser's runtime environment present constraints and complications that affect how site instances can be implemented and isolated from each other. For example, isolating site instances for security is desirable, but this is difficult to achieve without changing the behavior of web content. In this section, we show how the execution model for web programs suggests using a single thread and address space for all code within a site instance, and how the browser's trust model prevents site instances from being perfectly isolated.

2.3.1 Execution Model

Web program execution primarily involves two tasks: page rendering and script execution. These tasks are interleaved as an HTML page is parsed, and both tasks can result in DOM tree mutations. The browser presents a single threaded view of these events to avoid concurrent modifications to the DOM. This is partly necessary because JavaScript is a single threaded language with no concurrency primitives, and it suggests that a given page's rendering and script execution should be implemented on a single thread of execution.

Moreover, communicating pages within a site instance have full access to each other's DOM tree values and script functions. Because there are effectively no boundaries between these pages, it is important that no race conditions occur during their code's execution. We find that such race conditions are unfortunately possible in the Internet Explorer and Opera browsers, which both execute different windows in different threads, even if their pages can access each other.

To prevent concurrent DOM modifications and provide a simple memory model for shared DOM trees, we recommend that browser architectures place all web objects for a site instance, as well as their supporting browser components, into a single address space with a single thread of execution.

2.3.2 Trust Model

The trust model of the browser reveals the extent to which site instances can be isolated, because it defines how web objects can interact. We now consider how credentials can specialize a web program for a user, and how the browser tries to prevent the resulting confidential information from flowing into other web programs. We find that site instance boundaries alone are not sufficient for enforcing the browser's trust model, so we cannot yet rely on them for security.

Credentials Many web programs, such as mail or banking sites, are only useful when specialized for individual users. Browsers provide several credential mechanisms (e.g., cookies, client certificates) that allow sites to identify users on each request. As a result, these web programs may contain confidential information that should not leak to other web programs. We refer to the combination of a web program and the user's credentials as a *web principal*.

In most cases, sites give credentials to the browser after the user has authenticated (e.g., using a password). The main challenge is that browsers attach credentials to each request based solely on the destination of the request and not which web program instance is making the request. Thus, sites are usually unable to distinguish between a user's site instances on the server side. As a result, site instances from the same site can only be partly isolated in the browser; they have independent DOM trees but share credentials.

Information Flow Given that both web principals and other resources on the user's computer may contain confidential information, the browser's trust model must ensure that this information does not leak into the wrong hands. In terms of information flow, the browser must prevent any confidential information from flowing into an unauthorized site instance, because it places no restrictions on what information can flow out.

Specifically, a site instance can send information to any site, because the browser permits it to embed sub-resources from any site. A site instance can simply encode information in the HTTP requests for sub-resources, either as URL parameters, POST data, or in the URL path itself.

As a result, today's browsers must already take steps to prevent information from other web principals or from the *user principal* (i.e., the resources outside the browser to which the user has access) from flowing into a web principal.

To protect the user principal, the browser abstracts away the user's resources and denies access to local files and devices. These restrictions reflect the fact that web programs are not granted the same level of trust as installed applications and should not have the same privileges.

The browser faces a harder challenge to protect web principals. This is because site instances are free to embed sub-resources from any site. These sub-resources carry credentials based on their own origin, not the origin of their enclosing page (unlike the access control rules described in Section 2.2). Such sub-resources may therefore contain information to which the site instance should not have access. For example, if a user logs into a bank in one site instance and visits an untrusted page in another, the untrusted page

can request authenticated objects from the bank's site. This is an instance of the confused deputy problem [Hardy 1988], and it can lead to cross-site request forgery attacks for sites that do not defend against it [Watkins 2001].

In practice, today's browsers try to keep sub-resources opaque to their enclosing page, to prevent such information from flowing into the site instance. For example, script libraries can be executed but not directly read by a page's code, and a page's code cannot access the contents of binary objects like images. Pages also cannot transmit sub-resources back to their own site.

The consequence of this is that the browser must rely on subtle logic in its components, such as the script engine and DOM bindings, to maintain its trust model. Effectively isolating site instances is not sufficient, since a single site instance may include code or objects which its own pages should not be allowed to access.

2.4 Compromises for Compatibility

The practical constraints discussed above demonstrate the challenges for isolating web program instances in the browser in a compatible way. We conclude Section 2 by discussing the main compromises needed to maintain backwards compatibility, which is a critical goal for products like Chromium that seek adoption by real web users. These compromises permit compatibility but hold us back from perfectly isolating web program instances and their confidential information.

Coarse Granularity Our use of sites to define web programs represents one compromise. A site like `google.com` may host many logically separate applications (e.g., search, mail, etc.), perhaps even on different sub-domains. However, the architecture can only identify these as part of the `google.com` site. Any two pages from the same site instance could legally access each other, so we cannot isolate them a priori. Fortunately, this coarse granularity of web programs is offset by the fact that separate instances can be isolated. Thus, mail and map programs from the same site need not share fate, if the user opens them in separate browsing instances.

Imperfect Isolation A second compromise is that site instances cannot be fully isolated, for two reasons. First, a small subset of the DOM API is not subject to the Same Origin Policy, so site instances within a browsing instance are not entirely independent. This API is shown in Figure 4 and mainly allows a page to access properties of connected windows and frames, without reaching into the contents of their pages. Fortunately, it can be implemented without interacting with the DOM trees of different site instances, allowing them to be placed in different address spaces. Second, multiple site instances from the same site share credentials and other storage, such as cached objects. As a result, the browser's storage component can at best be partitioned among sites, not site instances.

Writable Properties	location, location.href
Readable Properties	closed, frames, history, length, opener, self, top, window
Callable Methods	blur, close, focus, postMessage, toString, history.back, history,forward, history.go, location.reload, location.replace, location.assign

Figure 4. Properties and methods of the `window` object that can be accessed by cross-site pages in the same browsing instance.

Sub-Resource Credentials Ideally, the information contained in web principals could be strongly isolated, but the browser's policies for sub-resources and credentials require another compromise. For example, the architecture could prevent one web principal from using another web principal's credentials when requesting sub-resources. This approach would reduce the importance of keeping sub-resources within a site instance opaque, since they would not contain another web principal's sensitive information. Unfortunately, since credentials are currently included based only on the destination of a request, this may break some mashup sites that depend on embedding authenticated sub-resources from other sites. Thus, we let the browser rely on components like the DOM bindings to keep these sub-resources opaque.

Overall, we find that compatibility does tie our hands, but we can still provide architectural improvements to browsers by making reasonable compromises. We choose a coarse granularity for web programs, we require shared credentials and a small API between site instances, and we limit our ambitions for securely isolating web principals.

3. Using Processes to Isolate Web Programs

In the previous section, we showed how the browser can be divided into independent instances of web programs, without sacrificing compatibility with existing web sites. However, most current browsers employ the monolithic architecture shown in Figure 1, combining all web program instances and browser components into a single operating system process. This leads to interference between web program instances in the form of fault tolerance, accountability, memory management, and performance.

In this section, we present a multi-process browser architecture that attempts to isolate these instances and their supporting browser components to significantly reduce program interference. We show how the Google Chrome team has implemented such an architecture (with some caveats) in the Chromium browser, and we discuss how the architecture can provide specific robustness benefits.

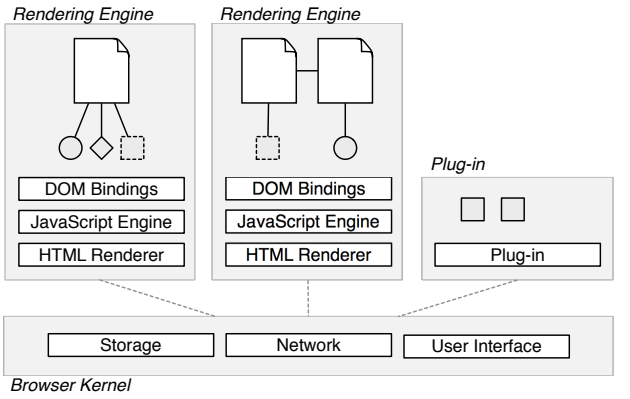


Figure 5. A multi-process architecture can isolate web program instances and their supporting browser components, from each other and from the rest of the browser. Plug-ins can also be isolated from the rest of the browser. Gray boxes indicate processes.

3.1 A Multi-Process Browser Architecture

We first show how to divide the browser's components between different operating system processes to isolate web program instances from each other. We choose OS processes as an isolation mechanism to leverage their independent address spaces and OS-supported scheduling and parallelism. We define three types of processes to isolate the browser's components and explain why certain components belong in each type. Our architecture is shown in Figure 5.

Rendering Engine We create a rendering engine process for each instance of a web program. This process contains the components that parse, render, and execute web programs. The HTML rendering logic belongs here, as it has no dependencies between instances and contains complex code that handles untrusted input. The DOM trees it produces similarly belong in this process, to keep the pages of a site instance together but those of different instances isolated. The JavaScript engine and the DOM bindings connecting it to the DOM trees are also placed here. These components share a single thread for rendering and script execution within a web program instance, but distinct instances can run on separate threads in separate processes. As we discuss in a separate report on Chromium's security architecture [Barth 2008], these processes can also be sandboxed to remove unnecessary privileges like filesystem access.

Browser Kernel We place most of the remaining browser components in a single process known as the browser kernel. All storage functionality (e.g., cookies, cache, history) is shared among instances of a web program and requires filesystem privileges, so it is placed in this process. The network stack is also privileged and can be shared among instances. Finally, the logic for managing the browser's user interface belongs in this process, as it is independent of the execution of web program instances.

Plug-ins We can introduce a third process type for running browser plug-ins like Adobe Flash, since plug-ins are effectively black boxes to the rest of the browser. Plug-in components could be loaded into each rendering engine process that needs them, but the Google Chrome team points out that this causes two problems [Google 2008c]. First, many plug-ins require privileges that are stripped from sandboxed rendering engine processes, such as filesystem access. Second, it would let a third party plug-in cause crashes in web program instances. Placing plug-ins in a separate process avoids these problems, preserving compatibility with existing plug-ins but losing potential security benefits of sandboxing them. Also, creating one plug-in process for each type of active plug-in avoids overhead from running multiple instances of a given plug-in component.

Using this architecture, instances of web programs can be safely run in separate processes from the rest of the browser without losing compatibility with existing content.

3.2 Implementation in Chromium

The Google Chrome team has implemented such an architecture in the Chromium web browser. Chromium's implementation isolates site instances using separate rendering engine processes. It also supports several other models for assigning web content to processes, based on the definitions from Section 2.2. We discuss these models below to reveal their implementation requirements and properties, and we then present caveats in Chromium's current implementation. Users can choose between the models with command line arguments, as documented online [Google 2008d].

- **Monolithic:** Chromium is designed to use separate processes for the browser kernel, rendering engines, and plug-ins, but it is capable of loading each of these components in a single process. This model acts as a baseline for comparisons, allowing users to evaluate differences between browser architectures without comparing implementations of different browsers.
- **Process-per-Browsing-Instance:** Chromium's simplest multi-process model creates a separate rendering engine process for each browsing instance. Implementing this model requires mapping each group of connected browser tabs to a rendering engine process. The browser kernel must display the process's output and forward user interactions to it, communicating via IPC. Frames can always be handled in the same process as their parent tab, since they necessarily share a browsing instance.

  A browsing instance, and thus a rendering engine process, is created when the user opens a new tab, and it grows when a page within the browsing instance creates a new connected tab. Chromium maintains the orthogonality between browsing instances and the visual group-

ings of tabs, allowing connected tabs to be dragged to different windows.

While this model is simple and provides isolation between pages in unconnected windows, it makes no effort to isolate content from different sites. For example, if the user navigates one tab of a browsing instance to a different web site than the other tabs, two unrelated web program instances will end up sharing the rendering engine process. This process model thus provides some robustness but does not isolate web program instances.

- **Process-per-Site-Instance:** The above process model can be refined to create a separate renderer process for each site instance, providing meaningful fate sharing and isolation for each web program instance. This model provides the best isolation benefits and is used in Chromium by default.

  To implement this, Chromium supports switching a tab from one rendering process to another if the user navigates it to a different site. Ideally, Chromium would strictly isolate site instances by also rendering frame pages in processes determined by their site. This is not yet supported, as we discuss in the caveats below.

- **Process-per-Site:** As a final but less robust model, Chromium can also consolidate all site instances from a site and simply isolate web programs, rather than web program instances. This model requires extra implementation effort to ensure that all pages from the same site are rendered by the same rendering engine process, and it provides less robustness by grouping more pages together. However, it still isolates sites from each other, and it may be useful in low resource environments where the overhead of additional processes is problematic, because it creates fewer rendering engine processes than process-per-site-instance.

Caveats Chromium's current implementation does not yet support strict site isolation in the latter two process models. There are several scenarios in which a single rendering engine process may host pages from multiple sites, although this could be changed in future versions.

This is partly because Chromium does not yet support the API permitted between windows showing different sites (from Figure 4), if those windows are rendered by different processes. We argue that supporting this API in Chromium is feasible, but until the support is implemented, Chromium should avoid breaking sites that may depend on the API. It does so by limiting the cases in which a tab is switched to a new rendering engine process during navigations, so that the API calls can complete within the same process. Specifically, it does not swap a tab's process for navigations initiated within a page, such as clicking links, submitting forms, or script redirects. Chromium only swaps processes on navigations via the browser kernel, such as using the location bar or bookmarks. In these cases, the user has expressed a more explicit intent to move away from the current page, so severing script connections between it and other pages in the same browsing instance is acceptable.

Similarly, Chromium does not yet render frames in a separate process from their parents. This would be problematic if secure isolation of site instances were a goal, but it is sufficient for achieving robustness goals.

Chromium also places a limit on the number of renderer processes it will create (usually 20), to avoid imposing too much overhead on the user's machine. This limit may be raised in future versions of the browser if RAM is plentiful. When the browser does reach this limit, existing rendering engine processes are re-used for new site instances.

Future versions could resolve these caveats to provide strict isolation of pages from different sites. Nonetheless, the current implementation can still provide substantial robustness benefits in the common case.

3.3 Robustness Benefits

A multi-process browser architecture can address a number of robustness problems that afflict current browsers, including fault tolerance, accountability, memory management, and performance issues. We discuss how the architecture is relevant for these problems, and how Chromium further leverages it for certain security benefits.

Fault Tolerance Faults in browser components or plug-ins are unfortunately common in these complex and evolving codebases, and they typically lead to a crash in the process where they occur. The impact of such a crash depends on both the browser architecture and which component is responsible.

In monolithic browsers, a crash in any component or plug-in will lead to the loss of the entire browser. To mitigate this, some browsers have added session restore features to reload lost pages on startup. However, users may still lose valuable data, such as purchase receipts or in-memory JavaScript state. Also, web programs that cause deterministic crashes may prevent such browsers from restoring any part of a session, since the browser will crash on each restore attempt. We have encountered this situation in practice.

In a multi-process architecture, many crashes can be confined to have much lower impact. Crashes in a rendering engine component, such as HTML renderers or JavaScript engines, will only cause the loss of one rendering engine process, leaving the rest of the browser and other web program instances usable. Depending on the process model in use, this process may include pages from a single site instance, site, or browsing instance. Similarly, plug-in components can be isolated to prevent plug-in crashes from taking the rest of the browser with them.

Note that crashes in browser kernel components, such as storage or the user interface, will still lead to the loss of the entire browser. However, in a related technical report [Barth 2008], we find that over the past year, the rendering engine

components of popular browsers contained more complexity and had twice as many vulnerabilities as the browser kernel components. Thus, tolerating failures in the rendering engine will likely provide much of the potential value.

Accountability As web pages evolve into programs, their demands for CPU, memory, and network resources grow. Thus, resource accounting within the browser becomes important for locating misbehaving programs that cause poor performance. In a monolithic browser, the user can typically only see resource usage for the entire browser, leaving him to guess which web program instances might be responsible. In multi-process architectures, resource usage for each process is available from the OS, allowing users to accurately diagnose which web program instances are to blame for unreasonable resource usage.

Memory Management Web program instances can also interfere with each other in monolithic browsers in terms of memory management. Browsers tend to be much longer-lived than web program instances, so a monolithic browser must use a single address space to execute many independent programs over its lifetime. Heavy workloads and memory leaks in these web program instances or in their supporting components can result in a large and fragmented heap. This can degrade performance and require large amounts of memory, and it can lead to large challenges for browser developers that seek to reduce memory requirements [Parmenter 2008]. In contrast, placing each web program instance and the components supporting it in a new process provides it a fresh address space, which can be disposed when the process exits. This simplifies memory reclamation and isolates the memory demands of web program instances.

Performance Performance is another concern as the code in web programs becomes more resource demanding. In monolithic browsers, a web program instance can interfere with the performance of both unrelated web programs and the browser itself. This occurs when separate instances are forced to compete for CPU time on a single thread, and also when the browser's UI thread can be blocked by web program actions, such as synchronous `XmlHttpRequests`. This is most evident in terms of responsiveness, where a large or misbehaving program instance can cause user-perceived delays in other web programs or the browser's UI. By isolating instances, a multi-process architecture can delegate performance and scheduling issues to the OS. This allows web program instances to run in parallel, improving responsiveness and taking advantage of multiple cores when available.

Security The browser's process architecture can also be used to help enforce security restrictions, although this is not the main focus of this paper. Monolithic architectures rely entirely on the logic in browser components, such as the DOM bindings, to enforce the trust model we discuss in Section 2.3.2. However, bugs may allow malicious web programs to bypass this logic, letting attackers install malware, steal files, or access other web principals.

In a recent report, we have shown how Chromium leverages its multi-process architecture to help enforce isolation between the user principal and web principals [Barth 2008]. Chromium uses sandboxes that restrict rendering engine processes from accessing the filesystem and other resources, which can help protect the user principal if a rendering engine is compromised.

It is also desirable to architecturally isolate web principals, to help prevent exploited rendering engines from stealing or modifying information in other web principals. As we discuss in Section 2.4, compatibility poses challenges for this because site instances can embed sub-resources from any site with the user's credentials. The architecture provides little benefit if confidential sub-resources are loaded by a malicious page in a compromised rendering engine. Thus, we leave a study of secure isolation of web principals to future work.

4. Evaluation

In this section, we evaluate the benefits and costs of moving from a monolithic to a multi-process architecture in the Chromium browser. We first demonstrate how this change improves robustness in terms of fault tolerance, accountability, and memory management. We then ask how the architecture impacts Chromium's performance, finding many advantages despite a small penalty for process creation. Finally, we quantify the memory overhead for the architecture and discuss how Chromium satisfies backwards compatibility.

Note that Chromium makes it possible to directly compare browser architectures by selecting between the monolithic mode and the process-per-site-instance mode using command line flags. We take this approach rather than comparing different web browsers to avoid capturing irrelevant implementation differences in our results.

Our experiments are conducted using Chromium 0.3.154 on a dual core 2.8 GHz Pentium D computer running Windows XP SP3. We use multi-core chips because they are becoming increasingly prevalent and can exploit the parallelism between independent web programs. All of the network requests in our tests were played back from disk from a previously recorded browsing session to avoid network variance in the results.

4.1 Is multi-process more robust?

We first use real examples to demonstrate that the multi-process architecture can improve Chromium's robustness.

Fault Tolerance We verified that crashes in certain browser components can be isolated in multi-process architectures. Chromium supports a built-in "`about:crash`" URL that simulates a fault in a rendering engine component. In monolithic mode, this causes the loss of the entire browser. In Chromium's multi-process architecture, only a single ren-

Workload	Monolithic	Multi-Process
Alone	9 (14)	4 (6)
With Top 5 Pages	1408 (2536)	6 (7)
With Gmail	3307 (5590)	6 (7)

Table 1. Average and worst case delay (in milliseconds) observed when interacting with a loaded page while other pages are loading or running.

dering engine process is lost. Any pages rendered by the process are simply replaced with an error message while the rest of the browser continues to function. Similarly, terminating a plug-in process in Chromium causes all plug-in instances (e.g., all Flash movies) to disappear, but the pages embedding them continue to operate.

Accountability Chromium's multi-process architecture allows it to provide a meaningful Task Manager displaying the CPU, memory, and network usage for each browser process. This helps users diagnose problematic site instances. In one real world scenario, we found that browsing two photo galleries caused the browser's memory use to grow considerably. In monolithic mode, Chromium's Task Manager could only report 140 MB of memory use for the entire browser, but in process-per-site-instance mode we could see that one gallery accounted for 104 MB and the other for 22 MB. This gave us enough information to close the problematic page.

Memory Management Many real web programs present heavy memory demands today, and leaks can occur in both web pages and browser components. For example, the image gallery mentioned above presents a heavy workload, while a recent bug in Chromium caused a rapid memory leak when users interacted in certain ways with news.gnome.org [Google 2008a]. In both cases, Chromium's multi-process architecture quickly reclaims the memory used by the offending site instance when its windows are closed, simply by discarding the process's address space. In contrast, the monolithic architecture continues to hold the allocated memory after offending windows are closed, only slowly releasing it over time as the browser's process attempts to free unused resources at a finer granularity.

4.2 Is multi-process faster?

A multi-process architecture can impact the performance of the browser because it allows site instances to run in parallel. We measure three aspects of the architecture's impact on Chromium's performance: the responsiveness of site instances, the speedups when multiple instances compute concurrently, and the latency introduced by process creation.

Responsiveness To test the browser's responsiveness, we look at the user-perceived latency for interacting with a page while other activity is occurring in the browser. Specifically, we inserted code into the browser kernel to measure the time interval between right-clicking on a page and the corresponding display of the context menu. This processing is handled by the same thread as rendering and script execution, so it can encounter contention as other pages load or compute. We perform a rapid automated series of 5 right clicks (500 ms apart) on a loaded blank page while other workloads are in progress. These workloads include loading the 5 most popular web pages according to Alexa [Alexa 2008], and loading an authenticated Gmail session. We report both the average and worst case latencies in Table 1, comparing Chromium's monolithic and multi-process architectures.

The monolithic architecture can encounter significant delays while other pages are loading or computing. Such "thread capture" may occur if the rendering and script computations performed by other pages prevent the active page from responding to events. The multi-process architecture completely masks these delays, keeping the focused page responsive despite background activity.

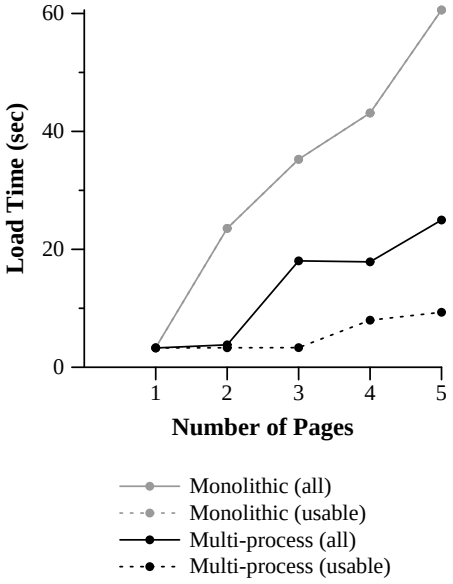
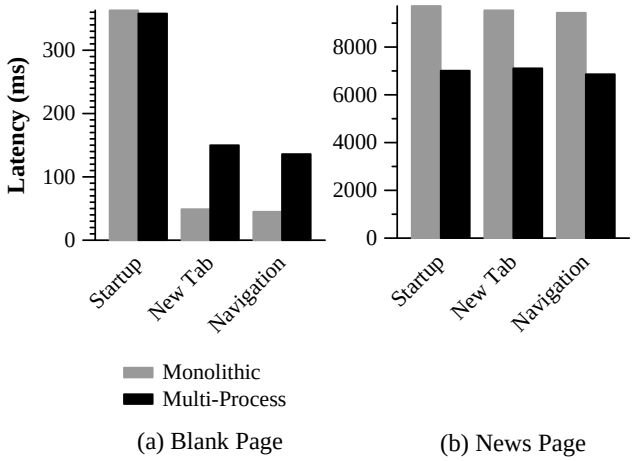


Figure 6. Solid lines show the total load time for restoring a session of realistic pages. Dotted lines show the time until at least one of the restored pages is usable. Both lines for the monolithic architecture overlap.

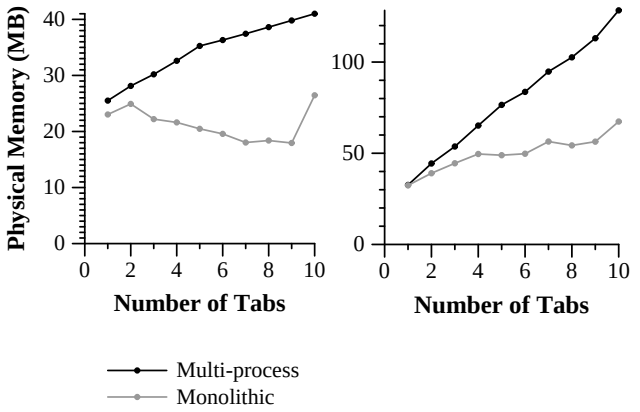
Speedup In some cases, multiple web program instances have enough pending work that considerable speedups are possible by parallelizing it, especially on today's multi-core computers. Speedups can occur when several pages are loading at once, or when pages in the background perform script operations while the user interacts with another page.

We consider the case of session restore, in which the browser starts up and immediately loads multiple pages from a previous session. Our test loads sessions containing between 1 and 5 Google Maps pages in different tabs, and we report the time taken for all pages to finish loading, based



(a) Blank Page (b) News Page

Figure 7. Latency for browser startup, tab creation, and a cross-site navigation, for both (a) a blank page and (b) a popular news page. Each task creates a process in the multi-process architecture.



(a) Blank Pages (b) Popular Pages

Figure 8. Memory overhead for loading additional blank or realistic pages in new tabs.

on measurements recorded within the browser. We compare results for Chromium's monolithic and multi-process architectures.

We also report the time until at least one of the pages becomes usable (i.e., responds to input events). In the monolithic case, this occurs when the last page finishes loading, because each loading page interferes with the others. For multi-process, we found we could interact with the first page as soon as it finished.

Our results are shown in Figure 6. On a dual core computer, the multi-process architecture can cut the time to fully load a session to less than half, and it allows users to begin interacting with pages substantially sooner.

Latency Although a multi-process browser architecture brings substantial benefits for responsiveness and speedups, it can also impose a latency cost for process creation. We use code in the browser to measure this latency for Chromium's monolithic and multi-process architectures in three common scenarios, each of which creates a process: starting the browser with a page, opening a page in a new tab, and navigating to a different site in a given tab. We present results for both a blank page and a popular news site, to put the additional latency in context.

Figure 7 shows the results. Startup times can actually improve for both pages in the multi-process architecture, because the browser kernel and rendering engine can initialize in parallel. For the blank page, new tab creation and cross-site navigations incur about a 100 ms penalty in the multi-process browser, due to process creation. As seen in Figure 7 (b), this is more than offset by speedups offered by the multi-process architecture. Such speedups are possible because the browser kernel must also perform computations to render a page, such as cache and network request management, which can run in parallel with the rendering engine.

4.3 What is the overhead for multi-process?

Moving to a multi-process browser architecture does incur a cost in terms of memory overhead. Each rendering engine process has its own copies of a set of browser components, causing the footprint of a new site instance to be larger than simply the footprint of the pages it contains.

Measuring this overhead poses a challenge because multiple browser processes may share large amounts of memory [Google 2008b]. We attempt to avoid double-counting this memory in our measurements. Specifically, the physical memory we report includes the private bytes and shareable (but not actually shared) bytes allocated by all browser processes, plus an approximation of the amount of shared memory. This approximation is the sum of each browser process's shared bytes divided by the number of processes.

Using this metric, we report the physical memory sizes of the monolithic and multi-process architectures after loading a number of pages in separate tabs. We consider the footprints of both blank pages and a set of the 10 most popular pages, as reported by Alexa.

Our results are shown in Figure 8. The blank page tests for the monolithic architecture show some variability, due to unrelated memory reclamations after browser startup. We confirmed this hypothesis by introducing page activity before running the tests, which eliminated the variability. We do not include this initial activity in our actual tests because it is biased against monolithic mode, which cannot reclaim the memory as easily as multi-process mode.

As expected, the multi-process architecture requires more memory per site instance than the monolithic architecture. For blank pages, the average site instance footprint rises from 0.38 MB to 1.7 MB between architectures. For popular

pages, it rises from 3.9 MB to 10.6 MB. The greater disparity in this case is likely due to caches and heaps that must be instantiated in each rendering engine as pages execute.

These footprints may vary widely in practice, depending on web program complexity. Many typical machines, though, have upwards of 1 GB of physical memory to dedicate to tabs, supporting a large number of average pages in either architecture: 96 in multi-process mode compared to 263 in monolithic mode. Also note that Chromium currently creates at most 20 rendering engine processes and then reuses existing processes, as discussed in Section 3.2.

4.4 Does multi-process remain compatible?

The multi-process architecture is designed to be compatible with existing web sites, as discussed in Section 2. Chromium's implementation also uses the WebKit rendering engine that is shared by Safari and other browsers, to avoid introducing a new engine for web publishers to test against. We are aware of no compatibility bugs for which the architecture, not the implementation, is responsible.

Nonetheless, both the architecture and Chromium's implementation exhibit some minor differences from monolithic browsers that may be observed in uncommon circumstances. Because of the shift from a global namespace for window names to a per-browsing-instance namespace, it is possible to have multiple windows with the same name. For example, the Pandora music site allows users to open a small player window. If a player window is already open, attempting to open another player from a second browsing instance will simply refresh the current player in monolithic browsers, but it will open a second player window in multi-process browsers. This is arguably a logical choice, as the user may consider the two browsing instances independent.

Chromium's implementation also does not yet support cross-process calls for the small JavaScript API that is permitted between page containers from different origins. As discussed in Section 3.2, Chromium attempts to keep such pages in the same process when they might try to communicate with this API. In practice, this is unlikely to affect many users, since most inter-window communication is likely to occur between pages from the same site.

5. Related Work

Existing browsers limit the script interactions between web program instances but face many robustness challenges without additional architectural support. Researchers have proposed some improved architectures, but they often do so at the expense of compatibility with existing content. Our work differs in that it automatically identifies and isolates web instances without disrupting existing content.

Monolithic Browsers Many existing web browsers use a monolithic architecture, which we have shown to be prone to robustness and performance problems. Some popular browsers (e.g., Firefox, Safari) run in a single process. Others, such as Internet Explorer (version 7 and earlier) and Konqueror, allow users to manually create new browser processes by starting a new instance of the browser. However, these processes contain more than a single browsing instance, and they make no attempt to isolate web program instances or particular browser components. Our prior technical report showed all of these browsers to be vulnerable to robustness problems due to their architecture [Reis 2007a].

Proposed Architectures Several research proposals have decomposed the browser into modular architectures to improve robustness and security. However, most have done so at the cost of compatibility by not identifying existing program boundaries. This makes them difficult to deploy on today's web, where pages might break without warning. For example, the OP browser isolates each web page instance using a set of processes for various browser components [Grier 2008]. The authors do not discuss communication or DOM interactions between pages of the same web program instance, which poses a challenge for this architecture. In Tahoma [Cox 2006], web programs are isolated in separate virtual machines. Program boundaries must be specified in manifests provided by the server, however, which are not available on today's web. SubOS tries to improve browser security with multiple processes, but the authors do not discuss the granularity of the process model nor the interactions between processes [Ioannidis 2001].

Recent beta releases of Internet Explorer 8 have introduced a multi-process architecture that can offer some of the same benefits as those discussed here [Zeigler 2008a;b]. IE8 separates browser and renderer components, and it runs renderers with limited privileges. However, IE8 does not distinguish or isolate site instances from each other. Instead, it ensures pages at different trust levels are isolated, such as local pages and internet pages. It also assigns pages to processes without regard to browsing instance or site instance boundaries. Thus, communicating pages from the same site may be placed in separate processes, exposing race conditions between their DOM modifications. We provide a separate contribution, identifying sites as web program boundaries and isolating site instances within the browser.

Site-Specific Browsers Several "site-specific browsers" have recently been introduced, sharing some of the same motivation as our work. These browsers treat certain web programs more like desktop applications, with desktop icons and fewer browser user interface features. Some site-specific browsers, such as Mozilla Prism and Fluid [Mozilla 2008, Ditchendorf 2008], run these web programs in separate processes to improve their isolation. However, these browsers require the user to explicitly create shortcuts for each web program of interest, providing no architectural benefit for any other sites the user visits. In contrast, our site instance abstraction makes it possible for browsers to automatically identify and isolate web program instances, without requiring input from the user.

6. Conclusion

The reliability problems in today's browsers are symptoms of an inadequate architecture, designed for a different workload than browsers currently face. We have shown that a multi-process architecture can address these reliability problems effectively by isolating instances of web programs and the browser components that support them. To do so, we have identified program abstractions within the browser while preserving compatibility with existing web content. These abstractions are useful not just for preventing interference between independent groups of web objects, but also for reasoning about the browser and its trust model. We hope that they can push forward discussions of future browser architectures and types of web content.

Our evaluation shows that Google's Chromium browser effectively implements such a multi-process architecture. It can be downloaded as Google Chrome from `http://www.google.com/chrome/`, and its source code is available at `http://dev.chromium.org/`.

Acknowledgments

This paper builds upon our earlier work with Brian Bershad and Henry M. Levy [Reis 2007a]. Our work relies heavily upon the Chromium implementation, built by the Google Chrome team, and we credit them for many of the insights that led to this work on compatibility constraints for multi-process browser architectures. We would like to specifically acknowledge Darin Fisher from the Chrome team, with whom one of the authors worked closely to develop the site instance isolation policy in Chromium. We also thank Adam Barth, Collin Jackson, Ian Hickson, and our shepherd Rebecca Isaacs for helpful comments on this paper.

This research was supported in part by the National Science Foundation under grants CNS-0132817, CNS-0430477, CNS-0627367, and by the Torode Family Endowed Career Development Professorship.

References

[Alexa 2008] Alexa. Alexa Web Search - Top 500. `http://www.alexa.com/site/ds/top_500`, 2008.

[Barth 2008] Adam Barth, Collin Jackson, Charles Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, Stanford University, 2008. `http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf`.

[Cox 2006] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.

[Ditchendorf 2008] Todd Ditchendorf. Fluid - Free Site Specific Browser for Mac OS X Leopard. `http://fluidapp.com/`, 2008.

[Google 2008a] Google. Issue 3666 - chromium - Tab crash (sad tab, aw snap) on jquery slidetoggle with -webkit-column-count greater than 1 - Google Code. `http://code.google.com/p/chromium/issues/detail?id=3666`, October 2008.

[Google 2008b] Google. Memory Usage Backgrounder (Chromium Developer Documentation). `http://dev.chromium.org/memory-usage-backgrounder`, 2008.

[Google 2008c] Google. Plugin Architecture (Chromium Developer Documentation). `http://dev.chromium.org/developers/design-documents/plugin-architecture`, 2008.

[Google 2008d] Google. Process Models (Chromium Developer Documentation). `http://dev.chromium.org/developers/design-documents/process-models`, 2008.

[Grier 2008] Chris Grier, Shuo Tang, and Samuel T. King. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*, 2008.

[Hardy 1988] Norm Hardy. The Confused Deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–8, October 1988.

[Hickson 2008] Ian Hickson and David Hyatt. HTML 5. `http://www.w3.org/html/wg/html5/`, October 2008.

[Ioannidis 2001] Sotiris Ioannidis and Steven M. Bellovin. Building a Secure Web Browser. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, June 2001.

[Mozilla 2007] Mozilla. Public Suffix List. `http://publicsuffix.org/`, 2007.

[Mozilla 2008] Mozilla. Prism. `https://developer.mozilla.org/en/Prism`, 2008.

[Parmenter 2008] Stuart Parmenter. Firefox 3 Memory Usage. `http://blog.pavlov.net/2008/03/11/firefox-3-memory-usage/`, March 2008.

[Reis 2007a] Charles Reis, Brian Bershad, Steven D. Gribble, and Henry M. Levy. Using Processes to Improve the Reliability of Browser-based Applications. Technical Report UW-CSE-2007-12-01, University of Washington, December 2007.

[Reis 2007b] Charles Reis, Steven D. Gribble, and Henry M. Levy. Architectural Principals for Safe Web Programs. In *HotNets-VI*, November 2007.

[Ruderman 2001] Jesse Ruderman. The Same Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, 2001.

[Watkins 2001] Peter Watkins. Cross-Site Request Forgeries. `http://www.tux.org/~peterw/csrf.txt`, 2001.

[Zeigler 2008a] Andy Zeigler. IE8 and Loosely-Coupled IE. `http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx`, March 2008.

[Zeigler 2008b] Andy Zeigler. IE8 and Reliability. `http://blogs.msdn.com/ie/archive/2008/07/28/ie8-and-reliability.aspx`, July 2008.