A Quantitative Evaluation of Privilege Separation in Web Browser Designs

Xinshu Dong, Hong Hu, Prateek Saxena, and Zhenkai Liang

Department of Computer Science, National University of Singapore
{xdong, huhong, prateeks, liangzk}@comp.nus.edu.sg

Abstract. Privilege separation is a fundamental security concept that has been used in designing many secure systems. A number of recent works propose re-designing web browsers with greater privilege separation for better security. In practice, however, privilege-separated designs require a fine balance between security benefits and other competing concerns, such as performance. In fact, performance overhead has been a main cause that prevents many privilege separation proposals from being adopted in real systems. In this paper, we develop a new measurement-driven methodology that *quantifies* security benefits and performance costs for a given privilege-separated browser design. Our measurements on a large corpus of web sites provide key insights on the security and performance implications of partitioning dimensions proposed in 9 recent browser designs. Our results also provide empirical guidelines to resolve several design decisions being debated in recent browser re-design efforts.

Keywords: Privilege separation, browser design, measurement

1 Introduction

Privilege separation is a fundamental concept for designing secure systems. It was first proposed by Saltzer et al. [31] and has been widely used in re-designing a large number of security-critical applications [9, 12, 28]. In contrast to a monolithic design, where a single flaw can expose all critical resources of a privileged authority, a privilege-separated design groups the components of a system into *partitions* isolated from each other. According to the principle of least privilege, each partition is assigned the minimum privileges it needs for its operation at run-time. Intuitively, this reduces the risk of compromising the whole system, because the attacker only gains a small subset of privileges afforded by the compromised component. Common intuition suggests that the more we isolate components, the better. We question this intuition from a pragmatic standpoint, and systematically measure the security benefits and costs of privilege-separating large-scale systems (such as a web browser) retroactively. Our empirical data suggests that "the more the better" premise is *not* categorically true. Instead, we advocate that practical designs may need to balance several trade-offs in retrofitting least privilege to web browsers.

Web browsers are the underlying execution platform shared between web applications. Given their importance in defeating threats from the web, web browsers have been a prime area where privilege separation is being applied. For instance, numerous clean-slate browser proposals [9,18,20,21,23,24,33,35] and commercial browsers like Bromium [2] and Invincea [3] are customizing privilege separation boundaries in web

browsers. However, excessive isolation between code components also incurs performance cost. Ideally, a practical browser design should balance security gains and the additional performance costs incurred by a new design. In browser design proposals, many important design dimensions are actively being debated. Should browsers put each web origin in its own partition? Should browsers host sub-resources (such as images, SVG, PDF, iframes) of a web page in separate partitions? Should sub-resources belonging to one origin be clubbed into the same partition? Should two code units (say, the JavaScript engine and the Document Object Model (DOM)) be assigned to different partitions? A systematic methodology to understand the empirical benefits and costs achieved by a partitioning strategy is important, but has not been investigated in depth.

Our study In this work, we study security and performance implications of choosing one or more of these partitioning dimensions in browser designs. To do this, we first extract a conceptual "blueprint" of the web browser that captures the logical components of a typical web browser. Then, we empirically measure a variety of parameters that measure security gains and performance costs of separating these logical components. This measurement is performed on a real web browser (Mozilla Firefox) using a large-scale test harness of the Alexa Top 100 web sites. Our measurements enable us to estimate the security benefits gained against the performance costs that arise when choosing a partitioning strategy.

Based on empirical data, we draw several inferences about the benefits and costs of design dimensions proposed in 9 recent browser design proposals. Our measurements lend pragmatic insights into some of the crucial design questions on how to partition web browsers. For example, we find that using separate OS processes to load cross-origin sub-resources requires 51 OS processes per web site, while giving marginal improvement in security for the increased performance cost. As another example, we find that isolating the JavaScript engine and the DOM creates a performance bottleneck, but also affords significant security gains. Many such empirical results are quantified in Section 5. Our measurements identify key performance bottlenecks in the browser designs we study, and we find that several of the bottlenecks identified correlate well with browser implementation efforts for design proposals that have public implementations. We hope our results and methodology serve as a baseline for further research on the problem, and are instructive in identifying early bottlenecks in upcoming browser designs.

Methodology Browsers are examples of large-scale systems, with millions of lines-of-code. For example, the browser we choose as the blueprint in this work (Firefox) has a development history of 8 years and comprises of over 3 million lines of code. If a security architect is tasked with privilege-separating an existing browser (like Firefox), how does she estimate security gains and performance bottlenecks of any particular privilege-partitioning configuration? In this paper, we take a step towards quantitatively studying this question with empirical data measurements. In previous research on privilege-separated browsers, performance measurements have been "after-the-fact", *i.e.*, after a chosen partitioning configuration has been implemented. In this work, we develop and report on a more rigorous measurement-based methodology that estimates the security benefits and performance costs, without requiring a time-intensive implementation. Our methodology precisely formulates back-of-the-envelope calculations that

security architects often use, and thereby systematizes a typical security argument with empirical rigor. Most prior works on browser re-design report performance on a small scale (typically on 5-10 sites). Our data-driven methodology leads to design decisions that are backed by a large-scale dataset.

Our methodology only aims to estimate weak upper bounds on the performance incurred by a proposed browser partitioning scheme. We recognize that these estimates can, of course, be reduced in actual implementations with careful optimizations and engineering tricks. However, our methodology lets us identify the likely bottlenecks where significant engineering effort needs to be invested. The metrics we evaluate in this work are *not* new and, in fact, we only systematize the measurement of quantities that prior works base their security arguments on. For instance, most prior works (somewhat informally) argue security based on two artifacts: (a) the reduction in size of the trusted computing base (TCB), and (b) the reduction in number of known vulnerabilities affecting the TCB after the re-design. To unify the security arguments previously proposed, we systematically measure these quantities using real-world data — 3 million lines of Firefox code and its corresponding bug database (comprising 8 years of Firefox development history).

Contributions Our goal in this paper is not to suggest new browser designs, or to undermine the importance of clean-slate designs and measurement methodologies proposed in prior work. On the contrary, without extensive prior work in applying privilege separation of real systems, the questions we ask in the paper would not be relevant. However, we argue to "quantify" the trade-offs of a privilege-separated design and enable a more systematic foundation for comparing designs.

In summary, we make the following contributions in this paper:

- We propose a systematic methodology to quantify security and performance parameters in privilege-separated designs, without requiring an implementation of the design.
- We perform a large-scale study on Firefox (>3 million LOC) on the Alexa Top 100 web sites.
- We draw inferences on the likely benefits and costs incurred by various partitioning choices proposed in 9 recent browser designs proposals, giving empirical data-driven insights on these actively debated questions.

2 Overview

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

2.1 Privilege Separation in Concept

Privilege separation aims to determine how to minimize the attacker's chances of obtaining unintended access to other part of the program. We consider each running instruction of a software program belongs to a *code unit* and a run-time *authority*. A code unit is a logical unit of program code, such as a software component, a function or a group of statements. The run-time *authority* can be a user ID or a web session, etc. Specifically, let p_i be the probability for any code unit or authority other than i to get unintended access to resources r_i belonging to i. From a purely security perspective, the

goal is to minimize the attacker's advantage. We can model this advantage using a variety of mathematical functions. For instance, an attacker's worst-case advantage from compromising a single vulnerability may be defined as $max(p_i)$; a privilege-separated design is good if it yields a large value of $(1 - max(p_i))$ [1]. However, as we argue in this paper, a practical privilege-separated design often departs significantly from this conceptual formulation. We argue that this purely security-focused viewpoint ignores the implicit performance costs associated with partitioning. Rather than focusing on mathematical modeling, we focus on the key methodology to quantify the benefits of a privilege partitioning scheme in this work.

2.2 Privilege Separation in Browsers

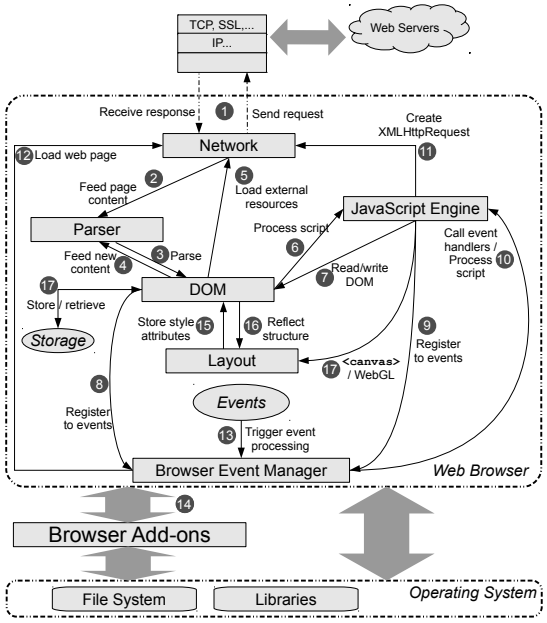


Fig. 1. Browser Blueprint. *It shows typical interactions between browser components in processing a web page.*

Blueprint To discuss trade-offs in partitioning, we use a conceptual blueprint that shows the various code units in a typical browser. We have manually extracted this from Mozilla Firefox, a popular web browser, and we show it in Figure 1[2]. We have confirmed that this conceptual blueprint is also consistent with WebKit-based browsers

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

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

and models sufficient details for comparing prior works on browser re-design. This blueprint intuitively explains the processing of web pages by various browser components. A web page is first received by the Network module that prepares content to be parsed by the HTML parser. The HTML parser creates a DOM, which can then invoke other execution engines such as the JavaScript engine, CSS, and so on. The legitimate flow of processed content between components is illustrated by arrows in Figure 1; for brevity, we skip explaining the details. In a single-process browser, all these components execute in the same partition. Web browser designs utilize privilege separation to isolate the resources owned by different authorities, which are defined next.

Isolating authorities Web browsers abstractly manage resources owned by one of the following authorities: *web origins*, the *system authority*, and the *user authority*. Web origins correspond to origins [4] of HTML pages, sourced scripts, CSS and so on. The system authority denotes the privilege of the browser core, also referred to as the chrome privilege. It has access to sensitive OS resources, such as the file system, network, display, etc. We associate the user authority to UI elements of the browser, which convey necessary security indicators to allow them to make sensible security decisions, such as security prompts, certificate warnings, access to preferences and settings [30].

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

- *CROSS-ORIGIN: Cross-Origin Data & Privilege Leakage*, due to vulnerabilities such as missing security checks for access to JavaScript objects or XMLHttpRequest status, and capability leaks [11].
- *WEB-TO-SYS: Web-to-System Privilege Escalation*, via vulnerable JavaScript APIs exposed by the browser components or plugins.
- *WEB-TO-COMP: Web-to-Component Privilege Escalation*, allowing attackers to run arbitrary code in vulnerable browser components, consisting of different memory corruption errors in the browser code.

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

- *USER: Confusion of User Authority*. These vulnerabilities may allow attackers to manipulate user interfaces to confuse, annoy, or trick users, hijacking their abilities in making reasonable security decisions. Recent incidents of mistakenly accepting bogus or compromised certificates [36] also belong to this category.
- *INTRA-ORIGIN: Intra-Web-Origin Data & Privilege Leakage*. This category of browser vulnerabilities results in running code within the authority of a web origin. These include bugs in parsing malformed HTML content, identifying charsets, providing HTTP semantics and so on. They can introduce popular forms of web attacks, such as XSS, CSRF and so on.

Partitioning dimensions 9 recent browser designs propose several ways of partitioning to mitigate the aforementioned threats. In this paper, we apply systematic methodology to study the security and performance trade-offs in these partitioning dimensions.

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

Table 1. Privilege Separation in Browsers *The table explains different partitioning dimensions in browser designs. For the right part of the table, same symbols denote the corresponding components are in the same partition.*

Table 1 summarizes the design dimensions considered in each browser design, and we explain these dimensions below.

- *By origin*: Each origin has a separate partition. This mitigates CROSS-ORIGIN vulnerabilities between web pages. For example, IBOS [33], Gazelle [35], Google Chrome [9], OP [20] and OP2 [21] all isolate primarily on origins [3]. In Chrome's default setting, web pages from different origins but belonging to the same "site instance"[4] are exceptions to this isolation rule.
- *By sub-resource*: When an origin is loaded as a sub-resource in another origin, say as an iframe or as an image, web browsers can isolate the sub-resources. This provides additional isolation between cross-origin resources, especially in mashups that integrate contents from various origins, and prevents CROSS-ORIGIN vulnerabilities from sub-sources explicitly included by an origin. For example, Gazelle [35] allocates a separate process for each destination origin of the resource and IBOS [33] uses a separate process for each unique pair of requester-destination origins; Chrome does not isolate sub-resources.
- *By component*: Different components are isolated in different partitions. Web browsers have proposed isolating individual components that are inadvertently exposed across origins, but do not need the full privileges of the system authority. For example, the OP browser [20] isolates the HTML parser and the JavaScript engine in different partitions. This prevents exploits of a WEB-TO-COMP vulnerabilities. Browsers also isolate components that need heavy access to resources of the system authority (such as the file system, network) from components that need only access to web origin resources. For example, Google Chrome [9] and Gazelle [35] separate components into web components (renderers) and system components (browser kernels). Parti-

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

[4] Connected web pages from the domains and subdomains with the same scheme. [17]

tioning along this dimension prevents WEB-TO-SYS vulnerabilities in the codebase of renderer partitions.

3 Quantifying Trade-offs with Empirical Measurements

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

3.1 Security Parameters

The goal of measuring security improvements is to estimate the reduction in the likelihood of an attacker obtaining access to certain privileged resources, which we introduced as probabilities p_i in Section 2.1. Estimating the resilience of software to unforeseen future has been an open problem [22, 25, 29]. In this work, our goal is not to investigate new metrics or compare with existing ones; instead, we aim to systematize measurements of metrics that have already been proposed in works on privilege separation. Security analysts argue improvements in security using two metrics: (a) reduction in TCB, i.e., the size of code that needs to be trusted to protect resource r_i, and (b) reduction in impact of previously known security vulnerabilities [5]. Next we explain the intuitive rationale behind the parameters we adopt in our evaluation. We leave details on how we measure them to Section 5.

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

S2: Severity weightage It is important to characterize the impact or severity of vulnerabilities. As we discuss in Section 2.2, different vulnerabilities give access to different resources. For instance, WEB-TO-SYS vulnerabilities give web attackers full access to system resources (including all other origins), so they are strictly more severe than CROSS-ORIGIN vulnerability. To measure this, we categorize security vulnerabilities according to their severity.

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

[5] Note that these metrics are instances of *reactive* security measurement, which have been debated to have both advantages [10] and disadvantages [29].

the relative complexity of various partitions and compute the size of TCB for different resources[6].

3.2 Performance Parameters

The precise performance costs of a privilege-separated design configuration can be precisely determined only after it has been implemented, because various engineering tricks can be used to eliminate or mitigate performance bottlenecks. However, implementing large re-designs has a substantial financial cost in practice. We propose a systematic methodology to calculate upper bounds on the performance costs of implementing a given partitioning configuration. These bounds are weak because they are calculated assuming a straightforward implementation strategy of isolating code units in separate containers (OS processes or VMs), tunneling all communications over inter-process calls as proposed in numerous previous works on browser re-design. This strategy does not discuss any engineering trick that can be used in the final implementation. We argue that such a baseline is still useful and worthy of systematic investigation. For instance, it lets the security analyst identify parts of the complex system that are going to be obvious performance bottlenecks. Our methodology is fairly intuitive and, in fact, often utilized by security architects in back-of-the-envelope calculations to estimate bottlenecks. We explain the performance cost parameters **C1-C7** we are able to quantitatively measure below. Mechanisms for measuring these parameters and the inference from combining them are discussed in Section 5.

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

C2: Size of data exchanged between code units If two code units are placed in separate partitions, read/write operations to data shared between them need to be mirrored into each partition. If the size of such data read or written is high, it may create a performance bottleneck. Two common engineering tricks can be used to reduce these bottlenecks: (a) using shared memory or (b) by re-designing the logic to minimize data sharing. Shared memory does not incur performance overhead, but has trades-off security to an extent. First, as multiple parties may write to the shared memory regions, it is subject to the time-of-check-to-time-of-use (TOCTTOU) attack [37]; second, complex data structures with deep levels of pointers are easily (sometimes carelessly) shared across partitions that makes sanitization of shared data error-prone and difficult to implement correctly. To estimate the size of inter-partition data exchange, we measure the size of data that are exchanged between different code units. This measurement identifies partition boundaries with light data exchange, where Unix domain sockets or pipes

[6] We do not argue whether code size is the right metric as compared to its alternatives [15, 26]; of course, these alternatives can be considered in the future. We merely point out that it has been widely used in previous systems design practice and in prior research on privilege separation.

are applicable, as well as boundaries with heavy data exchange where performance bottlenecks need to be resolved with careful engineering.

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

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

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

C6: Cost of an inter-partition call under different isolation primitives Partitioning the web browser into more than one container requires using different isolation primitives, such as processes and VMs. These mechanisms have different performance implications when they are applied to privilege separation. We measure the inter-partition communication costs of 3 isolation primitives in this work: Linux OS processes, LAN-connected hosts, and VMs; other primitives such as software-based isolation (heap isolation [8], SFI [34]) and hardware-based methods (using segmentation) can be calculated similarly.

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

4 Measurement Methodology

To measure the outlined parameters above, we take the following as inputs: *1)* an executable *binary* of a web browser with debug information, *2)* a blueprint of the browser, including a set of code units and authorities for partitioning, and *3)* a large test harness under which the web browser is subject to dynamic analysis.

We focus our measurements on the main browser components and we presently exclude measurements on browser add-ons and plugins. Our measurements are computed from data measured during the execution of the test harness dynamically, since computing these counts precisely using static analysis is difficult and does not account for runtime frequencies. Based on measurement data, we compare with partitioning choices in recent browser design proposals, and evaluate the security benefits and performance costs in those design dimensions.

In this work, we perform the measurement on a debug build of Firefox, a blueprint manually abstracted from Firefox and WebKit designs, historical Firefox vulnerabilities retrieved from Mozilla Security Advisories [27], and Alexa Top 100 web sites.

Since the engineering effort required to conduct such a large-scale study is non-trivial, we develop an assistance tool to automate our measurement and analysis to a large extent. Especially for the measurement of inter-partition function calls and data exchange sizes, we develop an Intel Pin tool. It applies dynamic instrumentation on the Firefox browser to intercept function calls and memory access. By maintaining a simulated call stack structure, we capture the caller-callee relationships during browser execution over test harness web pages. Before our experiments, we register accounts for the Alexa Top 100 web sites, when applicable, and log into these web sites using a vanilla Firefox browser under a test Firefox profile. Then we manually run Firefox instrumented by the Pin tool to browse the front pages of the web sites under the same test profile, so that contents requiring authentication are also rendered. As Firefox is slowed down by the Pin tool, it took one of the authors around 10 days to finish the browsing of the 100 web sites.

5 Experimental Evaluation

We conduct empirical measurements to obtain the data for evaluating browser designs. Our measurements are mainly conducted on a Dell™ server running Ubuntu 10.04 64bit, with 2 Xeon® 4-core E5640 2.67GHz CPUs and 48GB RAM. For the measurement of inter-partition communication overhead, we connected two Dell™ desktop machines with a dual-core i5-650 3.2GHz CPU and 4GB RAM via a 100 Mbps link.

5.1 Measurement Goals

Our measurements aim to measure the following:

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

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

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

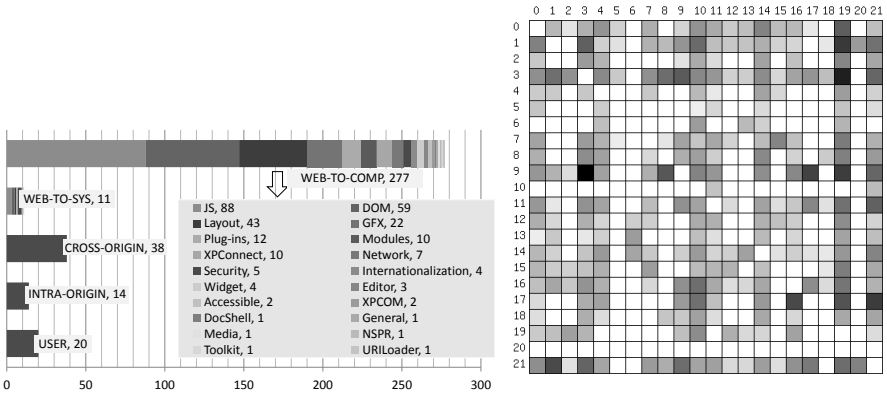
Comp#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
LOC	136	367	74	155	32	3	131	21	77	366	10	269	763	17	223	24	137	478	24	188	53

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

5.2 Measurement over Alexa Top 100 Web Sites

Next, we explain how we measure these metrics and present their results.

For *Goal 1*: security benefits. We measure the number of historical security vulnerabilities in each Firefox component according to each severity category (Security Param-



(a) Number of Historical Security Vulnerabilities in Firefox, Categorized by Severity and Firefox Components

(b) Gray-scaled Chart of Call Counts across Code Units. *Components are numbered as with Table 2.* Each cell at (i, j) denote the number of calls from Component i to Component j.

Fig. 2. Summary of Vulnerability Study and Performance Measurement

eters **S1, S2**) and the size of source code in Firefox components (Security Parameter **S3**).

We implement a Perl utility with 95 lines of code to crawl and fetch Firefox bug reports online [27]. According to the blueprint of browser components, and our classification of vulnerability severity, we count the 362 vulnerabilities[7] we have access to, by *1) browser component*, and *2) severity category*. Figure 2(a) depicts the number of Firefox vulnerabilities with our categorization outlined in Section 2.2. We can see that 76.5% of the security vulnerabilities are WEB-TO-COMP vulnerabilities (277), which can lead to code execution. There is also a large amount of CROSS-ORIGIN vulnerabilities (38), whereas the number of other categories is much smaller. Among browser components, the JavaScript engine has the largest number of vulnerabilities (88). The Layout module (43) and DOM (59) also have large amount of vulnerabilities. These are all major components consisting of complex browser logic. On the other hand, more peripheral components have less vulnerabilities. For example, the Editor has only 3 WEB-TO-COMP vulnerabilities. Such results are in line with our intuition that more complex and critical components tend to have more vulnerabilities discovered.

We use the `wc` utility to measure the lines of source code for all `.h, .c` and `.cpp` files in Firefox components. Table 2 lists the lines of source code we measure for different components in Firefox. Components such as JavaScript, Layout and Security, etc. have large size code size. These data reflect the (relative) complexity of different browser components (See **S3**).

For *Goal 2*: performance costs. We dynamically measure performance costs corresponding to Performance Parameters **C1-C5**, respectively.

Inter-code-unit call overhead. For Performance Parameters **C1** and **C2**, we apply our Pin tool on Firefox to browse Alexa Top 100 web sites, counting the number of

[7] 2 of them are uncategorized due to insufficient information.

function calls whose caller and callee belong to two different components, and the size of data exchanged during the function calls. We briefly discuss the results below, and the detailed measurement data can be found online at [1].

The numbers of inter-code-unit function calls (in 1000s) between different browser components are illustrated in Figure 2(b). These calls may become inter-partition calls after privilege separation. Thus, the larger the number is between the two components, the higher is the communication cost if they are isolated into different partitions. We find that there are 4,270,599,380 times of calls between the Layout engine and the DOM during our measurements, 369,305,460 times between the GFX rendering engine and the Layout engine, and 133,374,520 times between the JavaScript engine and the DOM. Heavy calls between these components correspond to tight interactions during run time, such as DOM scripting and sending layout data for rendering.

We also measure data exchange sizes between components. For example, the DOM and the Layout engine have larger data exchange than other components: 172,206.36 Kilobytes over the 100 web sites.

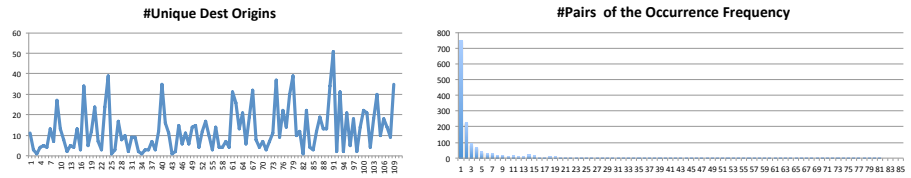
Cross-origin call overhead. Similarly, for calls and data exchange between different web origins (Performance Parameters **C3** and **C4**), we intercept the calls to client-side communication channels in Firefox, retrieve the caller and callee origins, and record the size of data passed in `postMessage` calls. For Performance Parameter **C5**, we intercept all network responses to Firefox, and identify whose requester and destination origins are different. We record such cases with the size of data passed in the HTTP response body. Table 3 summarizes the number of client-side calls to access other origins and the size of data exchanged in such calls.

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

Table 3. Cross-Origin Calls & Sub-Resource Loading

More results on sub-resource loading To evaluate in more detail the performance implications in using separate partitions for sub-resource loading, we measure the number of cross-origin sub-resources for each of the Alexa Top 100 web sites. Figure 3(a) illustrates the significant differences in the number of different origins of network sub-resource requests for each web page we measured. In our measurement, the largest number is 51, with `www.sina.com.cn`. Figure 3(b) shows that the reoccurrence rate of unique pairs of different requester and destination origins is very small. More than 746 pairs occur only once. In fact, there are in total 1,515 such unique pairs, averaged to 1,515 / 100 = 15 pairs for each page.

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



(a) Number of Different Destination Origins of Cross-Origin Resource Requests *The largest number of different destination origins from one site is 51, while the smallest number is 1.*

(b) Occurrence Frequencies of Unique Pairs of Different Requestor-Destination Origins *746 unique pairs only occur once, while only 164 unique pairs occur more than 15 times.*

Fig. 3. Sub-resource Loading Measurements

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

Table 4. Round-Trip Time (RTT) of Unix Domain Socket, Network and Cross-VM Communications, *in nanoseconds*, Averaged over 10,000 Runs Each

We use a simple client-server communication program to measure the inter-partition call costs between Unix domain sockets, between hosts connected via LAN, and between virtual machines on the same VM host. We average over 10,000 rounds of each primitive with message lengths varying from 50 to 8K bytes. Table 4 summarizes our measurements on round trip times for inter-partition communications with the three isolation primitives. Unix domain sockets are 6-10 times more efficient than cross-VM communications.

By checking the size of an empty process on different hosts, we estimate that the memory used by an almost-empty process is about 120k-140K. As this number stays very stable across different runs, we take this as the memory consumption of creating processes. For the Ubuntu guest OS we create, the writeable/private memory used by VirtualBox is about 25M bytes and the memory used by guest OS running in VirtualBox is about 90M bytes. We take 90M as the size of memory cost with a VM partition or a single host, and 25M as the memory overhead from a VM daemon in our quantification. Therefore, a Linux process incurs 90M / 130K = 709 times lower memory overhead than a VM.

5.3 Inference from Measurement Data

In this section, we summarize the high-level findings from our detailed measurements. Specifically, we revisit the partitioning dimensions outlined earlier and evaluate their security-performance trade-offs. We also summarize the performance bottlenecks that our measurements highlight.

Partitioning Dimension	#Vulnerabilities Migitated	Lines of Code Partitioned	Comm Cost	Data Exchanges Cost	Memory Cost
Single Process	0	N.A.	0	0	0.13K
One Process per Origin (w/o Cross-Origin Sub-Resource Isolation)	0	N.A.	0	0	130K
One Process per Origin (with Cross-Origin Sub-Resource Isolation)	38	N.A.	0.37ms	5.87MB	1.4MB
One Process per Pair of Requester-Destination of Sub-Resource	38	N.A.	0.91ms	7.19MB	2.1MB
Renderer/Browser Division	81	1,863K	2.59min	3.54MB	130KB
JS/DOM Separation (Process)	147	JS:367K DOM:155K	6.67s	572.6KB	130KB
JS/DOM Separation (Network)	147	JS:367K DOM:155K	3.78min	572.6KB	90MB
Layout/Window Manager (GFX+Widget) Separation	69	Layout:367K GFX+Widget:615K	19.15s	739.3KB	130KB
DOM/Layout Separation	102	DOM:155K Layout:367K	3.56min	1.68MB	130KB

Table 5. Security Benefits and Performance Costs of Partitioning Dimensions *Performance costs are per page, averaged over Alexa Top 100 web sites.*

Table 5 summarizes the estimated security benefits and performance costs for each design point along the dimensions being debated in present designs. The values in the table for performance costs are per web page, if applicable, averaged over the Top 100 Alexa pages.

Origin-based isolation One process per origin without separating cross-origin sub-resources have no security benefits. If contents from another origin hosted as sub-resources (such as PDF) can be processed in the same partition, security vulnerabilities can still permit unintended escalation of privileges. This is consistent with the observations made by several browser designs that propose hosting sub-resources in separate containers. Doing so, mitigates the CROSS-ORIGIN vulnerabilities (38 out of 362).

Sub-resource isolation Several browsers propose isolating each pair of requester-destination of sub-resources to be further isolated in separate partitions. Our data suggests that (a) this has no further security benefit in our model, and (b) it has a large performance cost. For instance, the memory cost of creating several partitions (using processes) is large and will be a performance bottleneck. In our measurement, one web page can include up to 51 third-party sub-resources. If all these cross-origin sub-resources are to be isolated by different processes, and consider a typical browser process need 20 Megabytes [5], then around 1 Gigabyte memory overhead will be incurred just for loading third-party resources for this single web page. Therefore, although sub-resource isolation can mitigate 38 CROSS-ORIGIN vulnerabilities, browsers may need to optimize memory usage for processes that load sub-resources before they can practically adopt this proposal.

It is interesting to compare our identified bottlenecks to choices made by today's web browsers. For instance, Google Chrome does not suffer from this performance bottleneck by making a security-performance trade-off. It adopts a different strategy by grouping resources according to a site-instance of the hosting page, which significantly reduces the number of processes created [17]. We leave the detailed definition and discussion of this strategy out of scope; however, we believe that our methodology does identify realistic practical constraints.

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

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

Our measurements identify potential performance bottlenecks that correlate with actual browser implementations. Specifically, we find that isolation between components in the renderer processes and the browser kernel process, as in Chrome, would incur very high performance overhead, such as between the GFX and the Layout engine. However, such performance bottlenecks do not appear in Chrome. Over the past few years, a substantial amount of efforts [7] have been spent on improving and securing the inter-partition communications in the Chrome browser. Besides, Chrome also uses GPU command buffers and other engineering tricks to improve performance of rendering and communication [16]. This verifies our observation that potential performance bottlenecks need to be re-engineered to reduce their overhead.

Component partitioning with high security benefits We identify a few browser components that have high security benefits to be isolated from other components. For example, the JavaScript engine is a fairly complex component with 367K lines of source code, has 88, i.e., 31.8% of, WEB-TO-COMP vulnerabilities. Isolating it from other browser components will mitigate a large faction of vulnerabilities. Other typical example components include the Layout engine with 367K lines of source code and 43 (15.5%) WEB-TO-COMP vulnerabilities, as well as GFX, the rendering component for Firefox, with 478K lines of source code and 22 (7.9%) WEB-TO-COMP vulnerabilities.

Component partitioning with high performance costs We identify the main browser components that have tight interactions with other browser components. Thus, isolating them from others would incur high performance costs. For example, our measurements find 133,374,520 function calls between the JavaScript engine and the DOM, and 369,305,460 calls between the GFX rendering engine and the Layout engine. To show why they can become performance bottlenecks, here is a simple calculation. Suppose they are separated by processes, a single RTT with Unix domain sockets would cost a delay of around 5000 nanoseconds. If there is no additional optimization is in

place, these numbers correspond to 133,374,520 * 5000 nanoseconds / 100 pages = 6.67 seconds/page and 18.47 seconds/page, respectively. Such performance overhead is prohibitively high. Security architects should either avoid such partitioning, or take further measures to optimize these performance bottlenecks.

6 Related Work

Privilege separation The concept of privilege separation in computer systems was proposed by Saltzer et al. [31]. Since then it has been used in the re-design of several legacy OS applications [12, 28] (including web browsers) and even web applications [5, 8, 19]. Similar to our goals in this work, several automated techniques have been developed to aid analysts to partition existing applications, such as PrivTrans [14], Jif/Split [38], and Wedge [13]. Most of these works have focused on the problem of privilege minimization, i.e., inferring partitions where maximum code executes in partitions with minimum or no privileges, while performance is measured "after-the-fact". Our work, in contrast, aims to quantify performance overhead with privilege-separated designs with only a blueprint without the actual implementations. Our work also differs with them by performing measurements on binary code, rather than source code.

Privilege separation in browsers Our work is closely related to the re-design of web browsers, which has been an active area of research [9, 18, 20, 21, 23, 24, 33, 35]. Our work is motivated by the design decisions that arise in partitioning web browsers, which performs a complex task of isolating users, origins and the system. Among them, IE uses tab-based isolation, Google Chrome [9] isolates web origins into different renderer processes, while Gazelle [35] further isolates sub-resources and plugins. Our measurements have shown that some web pages may include 51 sub-resources of different destination origins. Our data quantifies the number of partitions that may be created in such designs as well as in further partitioned browsers, such as OP [20] and OP2 [21]. In addition, our measurements also evaluate the performance costs in VM-based isolation, such as Tahoma [18], and memory consumption from separate network processes for sub-resources in IBOS [33] design. Our work advocates privilege-separated browsers for better security, and identifies potential performance bottlenecks that need to be optimized to trim their performance costs.

Evaluation metrics Estimation of security benefits using bug counts is one way of quantifying security. Riscorla et al. discuss potential drawbacks of such reactive measurement [29]. Other methods have been proposed, but are more heavy-weight and require detailed analysis of source code [22, 25, 32]. Performance measurement metrics such as inter-partition calls and data exchange have been identified in the design of isolation primitives such as SFI [34]. We provide an in-depth empirical analysis of these metrics in a widely used web browser (Mozilla Firefox).

7 Conclusion

In this paper, we propose a measurement-based methodology to quantify security benefits and performance costs of privilege-partitioned browser designs. With an assistance tool, we perform a large-scale study of 9 browser designs over Alexa Top 100 web sites. Our results provide empirical data on security and performance implications of various

partitioning dimensions adopted by recent browser designs. Our methodology will help evaluate performance overhead in designing future security mechanisms in browsers. We hope this will enable more privilege-separated browser designs to be adopted in practice.

Acknowledgments

We thank anonymous reviewers for their valuable feedback. This research is partially supported by the research grant R-252-000-519-112 from Ministry of Education, Singapore.

References

1. Additional tables on performance evaluation. http://compsec.comp.nus.edu.sg/bci/additional-tables.pdf
2. Bromium. http://www.bromium.com/
3. Invincea. http://www.invincea.com/
4. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium. CSF '10 (2010)
5. Akhawe, D., Saxena, P., Song, D.: Privilege separation in html5 applications. In: Proceedings of the 21st USENIX Security Symposium (2012)
6. Alexa: Top sites, retrieved in 2012. http://www.alexa.com/topsites
7. Azimuth Security: The chrome sandbox part 2 of 3: The IPC framework. http://blog.azimuthsecurity.com/2010/08/chrome-sandbox-part-2-of-3-ipc.html
8. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium. NDSS '10 (2010)
9. Barth, A., Jackson, C., Reis, C., The Google Chrome Team: The security architecture of the chromium browser. Tech. rep. (2008)
10. Barth, A., Rubinstein, B.I.P., Sundararajan, M., Mitchell, J.C., Song, D., Bartlett, P.L.: A learning-based approach to reactive security. In: Proceedings of the 14th International Conference on Financial Cryptography and Data Security. FC'10 (2010)
11. Barth, A., Weinberger, J., Song, D.: Cross-origin javascript capability leaks: detection, exploitation, and defense. In: Proceedings of the 18th USENIX Security Symposium (2009)
12. Bernstein, D.J.: Some thoughts on security after ten years of qmail 1.0. In: Proceedings of the 2007 ACM Workshop on Computer Security Architecture. CSAW '07 (2007)
13. Bittau, A., Marchenko, P., Handley, M., Karp, B.: Wedge: splitting applications into reduced-privilege compartments. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. NSDI'08 (2008)
14. Brumley, D., Song, D.: Privtrans: automatically partitioning programs for privilege separation. In: Proceedings of the 13th USENIX Security Symposium (2004)
15. Certification Authorities Software Team (CAST): What is a "decision" in application of modified condition/decision coverage (mc/dc) and decision coverage (dc)? http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf
16. Chromium: GPU command buffer. http://www.chromium.org/developers/design-documents/gpu-command-buffer

17. Chromium: Process models — process-per-site-instance. http://www.chromium.org/developers/design-documents/process-models#1_Process_per_Site_Instance
18. Cox, R.S., Gribble, S.D., Levy, H.M., Hansen, J.G.: A safety-oriented platform for web applications. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy (2006)
19. Felt, A.P., Finifter, M., Weinberger, J., Wagner, D.: Diesel: applying privilege separation to database access. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ASIACCS '11 (2011)
20. Grier, C., Tang, S., King, S.T.: Secure web browsing with the op web browser. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy (2008)
21. Grier, C., Tang, S., King, S.T.: Designing and implementing the op and op2 web browsers. ACM Transactions on the Web (2011)
22. Hart, T.E., Chechik, M., Lie, D.: Security benchmarking using partial verification. In: Proceedings of the 3rd USENIX Workshop on Hot Topics in Security. HotSec '08 (2008)
23. IEBlog: Tab isolation. http://blogs.msdn.com/b/ie/archive/2010/03/04/tab-isolation.aspx
24. Li, Z., Tang, Y., Cao, Y., Rastogi, V., Chen, Y., Liu, B., Sbisa, C.: Webshield: Enabling various web defense techniques without client side modifications. In: Proceedings of the Network and Distributed System Security Symposium. NDSS '11 (2011)
25. Lie, D., Satyanarayanan, M.: Quantifying the strength of security systems. In: Proceedings of the 2nd USENIX Workshop on Hot Topics in Security. HotSec '07 (2007)
26. McCabe, T.J.: A complexity measure. In: Proceedings of the 2nd International Conference on Software Engineering. ICSE '76 (1976)
27. Mozilla Foundation: Mozilla foundation security advisories. http://www.mozilla.org/security/announce/
28. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: Proceedings of the 12th USENIX Security Symposium (2003)
29. Rescorla, E.: Is finding security holes a good idea? IEEE Security and Privacy 3(1), 14–19 (Jan 2005)
30. Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H.J., Cowan, C.: User-driven access control: Rethinking permission granting in modern operating systems. In: Proceedings of the 2012 IEEE Symposium of Security and Privacy (2012)
31. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. In: Proceedings of the IEEE (1975)
32. Ta-Min, R., Litty L., Lie, D.: Splitting interfaces: Making trust between applications and operating systems. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. OSDI '06 (2006)
33. Tang, S., Mai, H., King, S.T.: Trust and protection in the illinois browser operating system. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI'10 (2010)
34. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. ACM SIGOPS Operating Systems Review 27(5), 203–216 (Dec 1993)
35. Wang, H.J., Grier, C., Moshchuk, A., King, S.T., Choudhury, P., Venter, H.: The multi-principal os construction of the gazelle web browser. In: Proceedings of the 18th USENIX Security Symposium (2009)
36. Wikipedia: DigiNotar. http://en.wikipedia.org/wiki/DigiNotar
37. Wikipedia: Time of check to time of use. http://en.wikipedia.org/wiki/Time_of_check_to_time_of_use
38. Zdancewic, S.A.: Programming languages for information security. Ph.D. thesis, Cornell University (2002)