The Web Interface Should Be Radically Refactored

John R. Douceur, Jon Howell, Bryan Parno
{johndo,howell,parno}@microsoft.com
Microsoft Research
Washington USA

Michael Walfish
mwalfish@cs.utexas.edu
Computer Science
University of Texas at Austin
Texas USA

Xi Xiong
xxx111@cse.psu.edu
Computer Science & Engineering
Penn State University
Pennsylvania USA

ABSTRACT

The Web API conflates two conflicting goals: serving developers by supporting a wide and growing suite of functionality, and providing applications with an isolated execution environment. We propose to split the API into two levels of interface: a low-level interface that governs the relationship between the application and the browser, and a set of high-level interfaces that govern the relationship between the application and its developer. We delineate a tiny set of properties needed by the low-level interface. We argue that this restructuring provides significant benefit to both developers and users.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/Server; Distributed applications; D.4.5 [Operating Systems]: Reliability—fault-tolerance

General Terms: Algorithms, Design, Experimentation, Performance, Reliability

Keywords: web procotol, programming interface

1 INTRODUCTION

A clean-slate refactoring of the web API can make web applications more robust and more useful. In fact, the very notion of a web API is misleading: It conflates the interface against which a developer writes a program with the interface that defines how a program is executed by a client. By separating these concepts, we enable more robust web clients and richer web applications without losing the essential properties that make the web applications attractive.

The essential properties of the web give users a fluid, carefree experience. Users confidently click on links to open unknown apps, an idea that would be foolish for desktop apps. Trying an app is easy, as is sharing or discarding it. Users don't update or patch apps; the apps retrieve their own code and content as needed. And notwithstanding these contrasts to the desktop, web apps are nearly as powerful and capable as desktop apps are.

While these properties are essential, the current web falls far short of achieving them optimally. The confidence with which users click on links is sometimes misplaced, as domain-isolation failures have been discovered and exploited in most major browsers [3]. And though it is easy to try a vanilla web app, many web apps require browser plug-ins, which in turn require updating and patching just like desktop applications. The functionality of web apps is limited by the web API's access restrictions. And web apps cannot incorporate the majority of the world's legacy code and libraries [10], slowing the rate of app development.

These problems arise from the conflation between two different interfaces: One is the interface against which a developer writes a program, which we call the *Developer Programming Interface* (DPI). The other is the interface that defines how a program is executed by a client, which we call the *Client Execution Interface* (CEI). The web API specifies a single interface layer, forcing these two interfaces—serving wildly different purposes—to be the same. Pressure to enrich the DPI conflicts with the desire to make the CEI robust.

We propose an alternative web architecture in which the CEI is as narrow and semantically simple as possible. Our proposed CEI provides an app with an isolated native-code execution environment [10] with primitive calls for memory allocation, thread management, synchronization, and an IP-layer network port. UI facilities are a raw frame buffer and events for keyboard and mouse input. To support the web's notion of links, there is a call for launching other applications. Other than these low-level primitives, all functionality is provided by libraries, each of which can export an arbitrary DPI to application code.

A small-CEI web is a big improvement over today's web. Developers can stick with a library that implements today's DPI (HTML, JavaScript, and a couple popular plug-ins), or they can substitute just about any other libraries and tools they wish (C#, Python, GTK+). A small, simple CEI frees developers from browser incompatibilities, so they can guarantee their users a consistent experience. A small, simple CEI also enables a far more robust browser, evicting all rich functionality out of the browser's core to the untrusted applications themselves. A developer's decision to incorporate a vulnerable library has no effect on other applications.

These are significant benefits, but achieving them requires addressing some important challenges. Our minimal CEI specifies the client's native instruction set, introducing an issue of cross-architecture compatibility. Our low-level CEI

Hotnets '11, November 14–15, 2011, Cambridge, MA, USA.

might curb opportunities for serendipitous extensions. A proliferation of DPIs may interfere with communication and interface consistency across applications. Functionality that is currently resident on the client will instead be downloaded as libraries, introducing performance concerns.

These challenges are readily addressable, leading to a system that will provide a better web experience than today's web. Users will run arbitrary apps without risking the integrity of their systems, and will never need to make security decisions about updating plug-ins. Developers will be free to choose among arbitrary components and tools, and will know their apps will behave consistently on users' systems. Others have attempted [5, 14, 20] to pry apart the web's DPI and CEI, but our clean-slate approach goes farther, constrained only to preserve the fundamental properties of web applications.

2 THE IRON PROPERTIES OF THE WEB

The web browser, once a simple viewer for static web pages, has evolved into an operating system for dynamic web applications. The diversity and capability of web applications have grown so dramatically that modern web applications rival the breadth and functionality of desktop applications. What has fueled this trend? Why do users find a browser to be a better application platform than a traditional desktop?

Somewhere within the amalgamation of standards and conventions that define 'web application', there must be some key properties that make such applications attractive. These important properties are unrelated to most of the de facto web API, such as HTML, DOM, CSS, JPEG, PNG and JavaScript. An entirely different set of web standards could be just as attractive and successful, as long as it retained a particular set of core properties.

Web apps are attractive because they are *isolated*, *rich*, *on-demand*, and *networked*:

- Isolated: Web applications cannot unilaterally affect other applications, so they are safe to try.
- Rich: Web applications are visually appealing, interactively responsive, and semantically powerful.
- On-demand: Web applications do not require installation or OS configuration, so they are easy to test drive and easy to point others to.
- Networked: Web applications make use of resources on the web, so they can access and integrate a growing and up-to-date set of disparate content.

These properties—which we call the IRON properties—are individually *necessary* to preserve the attractiveness of the current web. If web apps were not isolated or on-demand, the increased risk or burden of trying out a new app would reduce its rate of proliferation. If not networked, many interesting web apps (online maps, electronic commerce, cloud storage, etc.) could not function. A web of static pages, absent client-side execution, would not be very rich.

We further believe that these properties are jointly *sufficient* to provide the user experience that makes web applications attractive to users. To demonstrate this, we are constructing a *minimal* execution platform that satisfies the IRON properties. We aim to show that this platform can support all of today's web apps.

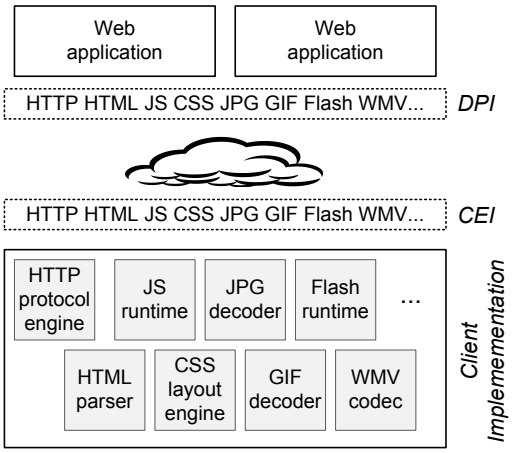

Figure 1—The CEI of the conventional web is closely bound to its DPI. A wide CEI puts much functionality on the client, where it impedes DPI innovation and hampers isolation.

3 WEAKNESSES OF THE CURRENT WEB API

Although the IRON properties are what makes web applications attractive, the current web API actually weakens all four of these properties.

Web apps are *not strongly isolated*, because the web API is very broad. Not only does the API include rendering interfaces for the suite of HTML standards (DOM, CSS) and an execution interface for JavaScript, but it also de facto includes interfaces to common image formats (GIF, JPEG, PNG) and popular plug-ins (Flash, JVM). This broad API is implemented via a large trusted computing base (TCB), which has evinced numerous security vulnerabilities and exploits [15], weakening isolation.

Web applications are undoubtedly rich, but far *less rich* than desktop applications. In large part, this is because desktop apps can build on a huge volume of existing code and libraries, written in arbitrary languages, and built using a wide variety of toolchains. By contrast, only a small fraction of legacy code is written in or translatable to a web-standard language such as JavaScript, Flash bytecode, or JVM bytecode [10].

The web API enables applications to be richer by taking advantage of plug-ins, such as ActiveX controls and runtimes for Flash and Java. However, this makes web apps *less on-demand*, because these plug-ins require updates and patches. Since the plug-ins are part of the TCB, this requires users to make configuration decisions: "Please install a new video codec" may be a quick route to malware infection. Attempts to use plug-ins to deploy entirely new runtimes, such as Silverlight, suffer from low uptake because they require user action beyond merely clicking on a link. A complex API leads to inevitable incompatibilities among browser implementa-

tions [11, 14], making the web less on-demand.

Web applications are *restrictively networked* by the web's same-origin policy (SOP), which restricts how a web app communicates with sites other than the app's origin. This policy inhibits interesting classes of networked applications, most notably peer-to-peer applications.

These weaknesses have been tackled before. Various research efforts have attempted to improve isolation [9, 17, 20, 21], provide native-code support [10, 23], and reduce the limitations of the SOP [1, 7]. However, the potential of these approaches is limited by the constraints of the current web API. Understandably, there is reluctance to change such a popular API, because of legacy concerns and because of developer familiarity with the existing API. However, we argue that the biggest problem is not so much changing the web API but rather disentangling two concepts that are conflated by the term "Application Programming Interface".

4 Deconstructing the web API

When a developer codes application-specific behavior into a desktop application, she builds on both libraries and OS system services. From her point of view, she writes the app against a *Developer Programming Interface* (DPI). When a client OS executes the application, it sees instructions that occasionally call OS services; it does not distinguish the app from its libraries. The client provides a *Client Execution Interface* (CEI). Typical desktop applications link dozens of libraries, making the DPI (e.g. GTK+) much higher-level than the CEI (e.g. Posix).

For web applications, the DPI and CEI are commonly identical[1]: The developer writes JavaScript or Flash, manipulates HTML and CSS, and presents JPEG or PNG images. Each of these features is implemented by the client browser, making the CEI equal to the DPI. See Fig. 1.

These two interfaces need not be identical; the CEI merely needs sufficient functionality to satisfy the IRON properties and to support code that implements the DPI. Otherwise, the CEI should be as small as possible: A small CEI makes applications more strongly isolated because it reduces the size of the shared TCB, thereby minimizing opportunities for security vulnerabilities across applications. A CEI that enables native-code execution, a la Xax [10] or Native Client [23], supports richer applications by allowing the reuse of extant code, libraries, and toolchains (see Fig. 2). With a small CEI, all rich code is outside the TCB, so applications can introduce new rich behavior without involving user configuration decisions, thus staying on-demand.

5 A minimal CEI for the web

In this section, we describe our specific proposal for a small, simple CEI, which we call *Zoog*. (See Fig. 3). We model the client as merely a tiny hosting center for applications: Every application is written in native-code, controlled entirely by the authoring vendor, and other applications may affect it only through network communication. This section describes the model in more detail, organized by how it addresses the IRON properties.

[1] Although toolkits such as GWT provide a higher-level DPI.

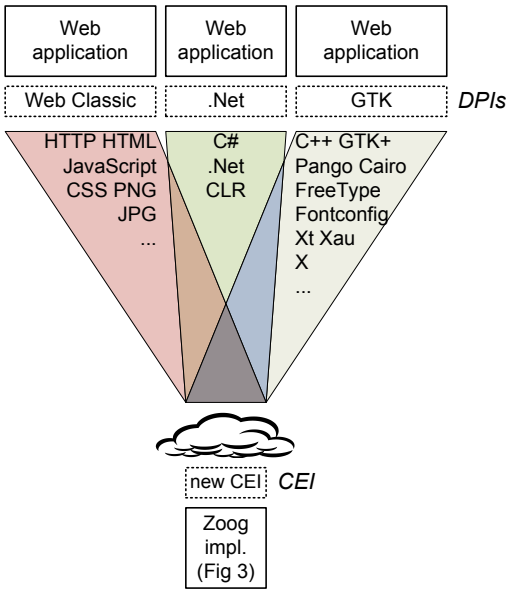

Figure 2—Explicitly decoupling CEI from DPI unbridles innovation among DPIs and admits a small Trusted Computing Base.

5.1 Support for Isolated Applications

In any system, the trusted computing base (TCB) is the portion that the designers are forced to assume is correct: In our context, bugs in the TCB compromise isolation. To minimize this danger, we want a small TCB, which implies that we also need a small CEI, since the CEI is implemented by the TCB. Indeed, the goal of TCB minimality pervades the design: Each CEI feature we introduce is both necessary to achieve IRON applications and essentially minimal.

The smallest TCB, and hence CEI, that we can imagine for isolating application execution is a *microkernel*. Microkernels can both isolate applications and give those applications access to low-level hardware features. The question that we answer in the subsections below is what types of calls the microkernel should expose to enable IRON applications.

5.2 Support for Rich Applications

In Zoog, application execution is expressed as a program in the client's native instruction set, running multiple threads in an isolated address space. This portion of the CEI requires about six calls for memory allocation, thread creation, and a futex-like scheduling primitive.

In support of rich user interaction, the CEI provides a low-level frame buffer. It also includes an overlapping window manager capable of labeling windows with application identity, providing a focusing gesture and feedback, and directing user keystrokes to the focused window. These are the simplest primitives necessary for the TCB to provide a path between the user and each application.

The display manager, which is also part of the TCB, may also interpret some user gestures (e.g., copy-paste, drag-and-drop) to convey cross-application user intent to the participating applications [18, 24]. Altogether, we anticipate about ten calls in support of display primitives.

5.3 Support for On-demand Applications

The web is on-demand because the user can click a link, and an app just appears. Today, the browser must identify the origin of each application, whether it was launched via link, opening a window, navigating the `window.url` property, or another method. Each additional mechanism complicates reasoning about isolation.

Zoog simplifies and secures application launch: One application launches another with a single call specifying an executable string of bits plus a cryptographic signature tying those bits to the launched application's identity. Most of the burden of launching is shifted to the participating applications: the launching app must fetch the launched app's boot loader,[2] and the launched app must bootstrap itself from the small boot loader.

The primary use of application identity in Zoog is to label windows to achieve trustworthy visual disambiguation [2]. The launch call strongly associates every app with a public key; a PKI maps keys to human-readable names which label windows. Visual disambiguation enables a user, before he types his bank password, to verify that he is typing into the bank's application, not a malicious app displaying the bank's logo. Our approach is a useful step, but the problem of dealing with visual ambiguity is deep and unsolved.

5.4 Support for Networked Applications

Any CEI that allows an unvetted application to communicate must prevent that connectivity from exploiting (1) other apps on the same host, (2) other hosts nearby on the network, and (3) other hosts across the internet.

For protecting apps on the same host, the current web API imposes complicated restrictions on inter-domain communication that admit subtle flaws [22]. By contrast, Zoog provides no cross-app communication other than a layer-3 network interface, which introduces seven CEI calls to support zero-copy packet I/O. All higher-layer network processing and message handling is the responsibility of a DPI library. Since the only way a malicious app can attack a target app is by sending packets, the case of another app on the same host reduces to the case of another machine on the internet.

For protecting other hosts nearby on the network, the web has evolved a complicated policy called the Same-Origin Policy (SOP). It allows an application to communicate with its origin server but disallows access to other servers, thereby preventing apps from exfiltrating data from delicate intranet servers to hosts outside the firewall. Because the pure SOP is too restrictive for many applications, there are exceptions

[2]The launching app and the launched app's server must agree on a boot-block fetch protocol, but that's outside the CEI and can evolve without updating clients.

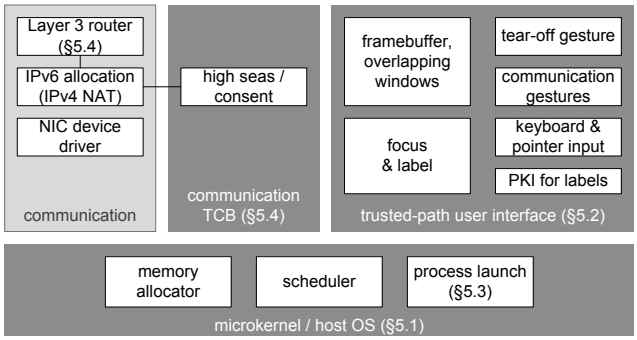

Figure 3—The Zoog CEI design is guided by the shape it imposes on the client implementation, shown here. Dark regions are part of the safety TCB; vulnerabilities in these modules can compromise the safety of every application. Light regions also implement the CEI, but failures only affect liveness: A failure in the network stack is no worse than a failure of the next-hop router.

for displaying images and executing scripts from any site, although not for reading the image or script content. The reasonableness of these exceptions is called into question by standards such as JSON that encode data as script, specifically working around SOP restrictions.

Because of these complex exceptions, it is not possible to implement the SOP in a browser without incorporating very high-layer knowledge about HTTP, MIME, JavaScript, the DOM, and even image decoding. Since this would vastly complicate the CEI, Zoog takes a radically different approach: Each application's IP pipe is connected logically outside any firewall; therefore, there are no other host nearby on the network. To implement this arrangement, the web client requires a mechanism to ensure that packets are routed to outside the firewall. We envision the client acquiring network access from a modified DHCP server or internet proxy that vouches its connection is high-seas. Any app that needs access to firewalled enterprise resources establishes its own secured connection from logical outside, analogous to the use of a VPN from a remote host.

For protecting other hosts on the internet, the threat is not in packet content but in packet volume: An attacker can lever many clients' resources to mount a DDoS attack. To protect against this threat, a Zoog client enforces correct source addresses in outgoing packets, and it can also employ source-side throttling mechanisms [8].

6 Benefits

6.1 A Clean CEI Frees Developers

Today's web DPI restricts developers to the HTML/CSS/JavaScript stack or a few plug-in alternatives, such as Flash or Java. The developer must test on four different brands of browser, as the wide interface is not consistently implemented. With Zoog, a developer can stick with this same DPI, but need only test against the one browser library incorporated into the application.

Even better, developers are free to program against other

DPIs, such as C#/.Net or C++/GTK/X. Technologies and toolchains may evolve independently and compete exclusively on their merits to developers. For instance, a developer can target a new version of Flash without waiting for the Flash runtime to be adopted explicitly by end users.

6.2 A Clean CEI Improves Robustness

The Zoog CEI pushes innovation from the client to the developer's side of the interface. This philosophy keeps the TCB small and evolving slowly. Prior research [6, 13, 19] suggests that the components in §5 might be implemented so compactly that formal verification is feasible [12].

A robust TCB fulfills the promise of isolation, that users need not worry whether it is safe to click a link. Users will still face authorization decisions, but at higher levels of abstraction, inside applications.

6.3 A Clean CEI Improves Predictability

Strong isolation offers predictability to developers. An app will not be affected by the presence of other applications on a client machine, because the CEI provides no opportunity for the user to approve a unilateral violation of isolation. Instead, the developer can be certain that any interactions with outside applications occur through channels that the app explicitly supports.

7 Considerations

This proposal changes a number of aspects of today's web, raising questions to answer and issues to consider.

7.1 Cross-Architecture Compatibility

Being minimal and low-level, our proposed CEI appears to lack a benefit of today's web API: Zoog assumes that applications provide executable code in the client machine's instruction set and thus has a binary format dependency.

This problem can be addressed with several mechanisms of competing merits. For example, vendors starting with C or C++ applications can compile for several architectures. A developer writing code in a managed language (Java, C#, Flash, JavaScript) can depend on the runtime supplier to transparently provide an appropriate architecture-specific implementations. LLVM can bring the same benefit even to C/C++ sources [23]. Binary rewriting works without help from the compiler toolchain.

The problem must be addressed, but by leaving its solution to a layer above the web's CEI, the CEI itself stays cleaner and exerts fewer constraints on the developer.

7.2 Mashups and Serendipitous Extension

A benefit of today's open web API is that third parties can arrive, inspect the client-server traffic, and create a new application or service that interacts with the original. In Zoog, the CEI no longer forces specific client-server protocols. We doubt this will ruin serendipitous enhancement. The few applications that would obfuscate in Zoog are obfuscating today in JavaScript. The majority of applications that stumble into standard serialization protocols (XML, JSON) included in today's DPI will continue to do the same or an equivalent (e.g., Python pickles or Java RMI).

7.3 Navigation and History

Today's browser is so named because it provides the user model of navigating through pages of disparate content, with facilities like history and coloring of visited links. In Zoog, navigation facilities are encoded as a protocol between participating applications, not specified by the CEI. A parent tells the child its identity, and when the user clicks a link in the child, the child asks the parent to destroy it and launch the new app in its place. Another protocol lets a user-trusted history aggregator render links for participating applications. Apps agree voluntarily to such protocols and to the degrees of information they share with other apps.

7.4 Adversarial Vendor Relationships

Who has the final say about how applications behave? The conventional answer is the user: She can flip a browser privacy setting, or install AdBlockPlus or NoScript, to control application behavior even when it is adversarial. Zoog takes the opposite stance, giving developers the final say.

The merits of the conventional approach are well-understood; what are the costs? When application behavior is ultimately determined by the user, then when that configuration breaks, only the user can repair it. A naïve user installing an extension may not anticipate or diagnose its negative interactions with other applications.

We hope that most vendors will behave non-adversarially. For exceptions, a third party can offer a "wrapper" service that interposes on the adversarial vendor. This is similar to the sort of system-wide extension a user can install today, but it affects only a single application. More importantly, its presence is clear to the user, because the user must provide the interposing service with his credentials for the adversarial service. We hope this escape valve is used rarely, since it erodes the benefits of developer predictability (§6.3).

7.5 Cross-Application Communication

Although the foundation of an IRON platform is isolation, applications will want to interact, as today's desktop software does. We wish to enable this interaction without widening the CEI and eroding isolation.

Since Zoog provides no communication other than network communication, we imagine reinventing conventional desktop application communication patterns as network protocols. Each application may independently decide whether to accept the risk of implementing a given protocol, and it may select the protocol implementation that best balances functionality against security.

7.6 Performance

The Zoog CEI seems to imply that every web page must download a whole browser. However, we anticipate that there

will be only a few common runtime stacks across most applications. By using content-based naming, a cache of these components can be shared while isolating execution [4]. Any app that uses an uncommon stack will incur an initially long load time, but this is far less of a penalty than an app faces today if it relies on an uncommon plug-in.

8 Related Work

Java was an early purpose-built CEI. However, it was not very small, including the JVM and a growing set of security-sensitive libraries. Java also restricted developers to its own toolchain, which took years to mature.

The first project to propose a narrower CEI for today's web was Tahoma [5], which suggested a hardware virtual machine (VM) interface for web applications. Tahoma admitted arbitrary DPIs, but made no effort to characterize a minimal CEI. The Tahoma paper is mute on whether to preserve the SOP (which would confound the architecture's simple isolation story) or somehow replace it.

Xax [10] and NaCl [23] demonstrate that rich, desktop-evolved DPIs can run on very small CEIs. These projects make no effort to identify a minimal CEI; instead, they expand the web CEI with yet another content container. Drawbridge [16] makes no claim to reshape the web, but shows a very rich DPI (Windows) running on a small CEI. We consider these projects evidence that one well-chosen narrow CEI admits a wide variety of DPIs.

Another school of research [9, 17, 20, 21] refactors the browser into a trusted kernel and untrusted rendering components, narrowing the TCB while keeping the conventional web DPI. This TCB could evolve into a refactored CEI as implementations admit new DPI code above their TCBs. However, to remain strictly web-compatible, a CEI must enforce convoluted rules (§5.4) that cross software layers (networking, script execution, font rendering, etc.), which limits how small such a TCB may become. We advocate a much more severe refactoring that admits a tighter CEI and admits a broader variety of DPIs.

Chrome is a good example of the difficulty of cleaning up the web CEI abstraction: Chrome defends the host machine with a strong outer boundary. Protection between adversarial web principals within the same tab and against attacks exploiting the host's network position (SOP) come only from a weaker "inner" boundary whose design is contaminated by the layer-crossing violations of the web's CEI.

Atlantis also separates DPI from CEI [14]. It uses a microkernel analogy to describe its CEI, and then reconstructs the complexity of existing browsers' rendering components above the CEI. The Atlantis CEI, however, uses a high-level computation model (an AST-based logical machine) and high-level UI primitives (`renderText`, `renderWidget`). Atlantis allows extension of the browser-like DPI, but cannot support diverse runtimes (e.g., C#, Java, C++, Python).

In contrast with these proposals, we argue for a CEI (§5) smaller and much closer to the raw hardware. This choice both enables implementations with smaller TCBs, and a admits a wider variety of DPIs. This choice is only feasible if we also abandon the web SOP; our high-seas networking (§5.4) is a novel alternative that solves the same problems without introducing high-level abstractions into the CEI.

9 Summary and conclusion

The web is attractive as an application platform because it satisfies the IRON properties. By separating the web's Developer Programming Interface from its Client Execution Interface and making the latter as small as possible, we can enable developers to choose arbitrary toolchains, and we can improve the robustness and predictability of web applications.

References

[1] Adobe. Cross-domain policy file specification. `http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html`.

[2] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symp. on Security and Privacy*, 2007.

[3] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *ACM Computer and Communications Security*, 2007.

[4] C. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa. Slinky: static linking reloaded. In *USENIX ATC*, 2005.

[5] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for Web applications. In *IEEE Symp. on Security and Privacy*, 2006.

[6] N. Feske and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *IEEE ACSAC*, 2005.

[7] I. Fette. The WebSocket protocol. `http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol`, 2011.

[8] J. M. Gregory, G. Prier, and P. Reiher. Attacking DDoS at the source. In *IEEE ICNP*, 2002.

[9] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symp. on Security and Privacy*, 2008.

[10] J. Howell, J. R. Douceur, J. Elson, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *USENIX OSDI*, 2008.

[11] E. Kıcıman and B. Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *ACM SOSP*, 2007.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *ACM SOSP*, 2009.

[13] J. Liedtke. Toward real microkernels. *CACM*, 39(9):70–77, Sept. 1996.

[14] J. Mickens and M. Dhawan. Atlantis: Robust, extensible execution environments for Web applications. In *ACM SOSP*, 2011.

[15] NIST Vulnerability Database. http://nvd.nist.gov/nvd.cfm.

[16] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *ACM ASPLOS*, 2011.

[17] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *ACM EuroSys*, 2009.

[18] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. Technical Report MSR-TR-2011-91, Microsoft Research, Aug. 2011.

[19] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS Trusted Window System. In *USENIX Security Symposium*, 2004.

[20] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *USENIX OSDI*, 2010.

[21] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security Symposium*, 2009.

[22] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I still know what you visited last summer: User interaction and side-channel attacks on browsing history. In *IEEE Symp. on Security and Privacy*, 2011.

[23] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symp. on Security and Privacy*, 2009.

[24] K.-P. Yee. Aligning Security and Usability. *IEEE Security and Privacy*, 2(5):48–55, September 2004.