Protection and Communication Abstractions for Web Browsers in MashupOS

Helen J. Wang
Microsoft Research
Redmond, WA, USA
helenw@microsoft.com

Xiaofeng Fan
Microsoft Research
Redmond, WA, USA
xiaoffan@microsoft.com

Jon Howell
Microsoft Research
Redmond, WA, USA
howell@microsoft.com

Collin Jackson
Stanford University
Palo Alto, CA, USA
collinj@cs.stanford.edu

ABSTRACT

Web browsers have evolved from a single-principal platform on which one site is browsed at a time into a multi-principal platform on which data and code from mutually distrusting sites interact programmatically in a single page at the browser. Today's "Web 2.0" applications (or *mashups*) offer rich services, rivaling those of desktop PCs. However, the protection and communication abstractions offered by today's browsers remain suitable only for a single-principal system—either *no trust* through complete isolation between principals (sites) or *full trust* by incorporating third party code as libraries. In this paper, we address this deficiency by identifying and designing the missing abstractions needed for a browser-based multi-principal platform. We have designed our abstractions to be backward compatible and easily adoptable. We have built a prototype system that realizes almost all of our abstractions and their associated properties. Our evaluation shows that our abstractions make it easy to build more secure and robust client-side Web mashups and can be easily implemented with negligible performance overhead.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Security, Standardization

Keywords

Browser, Web, same-origin policy, protection, communications, security, multi-principal OS, abstractions

1. INTRODUCTION

Web browsers are becoming the single stop for everyone's computing needs including information access, personal communications, office tasks, and e-commerce. Today's Web applications synthesize the world of data and code, offering rich services through Web browsers and rivaling those of desktop PCs. Browsers have evolved to be a multi-principal operating environment where mutually distrusting Web sites (as principals) interact programmatically in a single page on the client side, sharing the underlying browser resources. This resembles the PC operating environment where mutually distrusting users share host resources.

However, unlike PCs that utilize multi-user operating systems for resource sharing, protection, and management, today's browsers do not employ any operating system abstractions, but provide just a limited all-or-nothing trust model and protection abstractions suitable only for a single-principal system: There is either *no trust* across principals through complete isolation or *full trust* through incorporating third party code as libraries. Consequently, Web programmers are forced to make tradeoffs between security and functionality, and oftentimes sacrifice security for functionality.

In the MashupOS project, we aim to design and build a browser-based multi-principal operating system. Among the myriad of operating system issues, we focus on the most imminent needs of today's browsers: abstractions for protection and communication. The goal of protection is to prevent one principal from compromising the confidentiality and integrity of other principals, while communication allows them to interact in a controlled manner.

We follow the principles below in designing our abstractions for Web programmers:

- *Match all common trust levels*: We must understand all the common trust levels between Web content providers and integrators and aim to provide abstractions matching these levels of trust. Otherwise, Web programmers would face making tradeoffs among trust levels, either trusting more, and sacrificing security, or trusting less, and losing functionality.
- *Strike a balance between ease-of-use and security*: One might argue that a system is either secure or insecure and there should be no middle ground. This may be true for designing a security system like an authentication system, but not true when designing abstractions for programmers. A rigid set of abstractions that tie programmers' hands and limit flexibility for the purpose of better security often has a short life span, since programmers can build up libraries on the rigid interfaces to make their lives easy, resulting in a de facto abstraction for all programmers. It would be better for abstraction designers to design those abstractions with security in mind. Our goal here is to provide a full set of abstractions and enable programmers to build robust and secure services that match their trust expectations (see above bullet), rather than to make it impossible for programmers to shoot themselves in the foot.
- *Easy adoption and no unintended behaviors*: Allowing easy adoption is paramount in our abstraction design. We must

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

ensure that our abstractions allow programmers to provide fallback mechanisms when Web pages using them are rendered by legacy browsers. We also must ensure that there are no undesirable interactions between new services and old services in the new browser environment where our abstractions are supported.

By analyzing the trust relationship between content providers and integrators, we identify four types of content that require support from the Web and browsers: (1) *isolated content* that is intended to be completely isolated from other sites (domains), (2) *access-controlled content* that is isolated but allows message passing across domains to give mediated access to the content, (3) *open content* that allows any domain to access and integrate as the domain's own content, and (4) *unauthorized content* that assumes no privileges of any domain. Existing browser abstractions support only isolated content with the <frame> abstraction and open content with the <script> abstraction, resulting in an all-or-nothing trust model.

We propose abstractions for the missing content types and trust relationships. We advocate *unauthorized content* to be a fundamental addition to today's Web content provisioning. We introduce <Sandbox> and <OpenSandbox> abstractions and a provider-browser protocol to enable content providers to publish and integrators to consume unauthorized content without liability and overtrusting, providing both security and ease in creating client mashups. Such support can also fundamentally combat Cross Site Scripting attacks (a prominent threat in today's Web and a consequence of the all-or-nothing trust model) while allowing the richest possible third party content. We have also proposed the <ServiceInstance> abstraction for isolation, fault containment, and as the unit of resource allocation and *CommRequest* for cross-domain communications unifying recent proposals.

We have designed these new abstractions to be backward compatible and free of undesirable interactions with legacy browsers and legacy content. Our prototype implementation and evaluation demonstrate that the abstractions can be practically integrated into modern browser software with negligible overhead.

For the rest of the paper, we first give background in Section 2. In Section 3, we describe browser resources and define MashupOS principals that own these resources. We analyze trust relationships between content providers and integrators and identify missing content types, trust relationships, and abstractions in Section 4. In Section 5, we describe unauthorized content in detail and our sandbox abstractions for it. We present our <ServiceInstance> abstraction in Section 6 and the *CommRequest* abstraction in Section 7. In Section 8, we show how our sandbox and ServiceInstance abstractions can be utilized to combat Cross Site Scripting attacks. We present our Internet Explorer-based prototype in Section 9. In Section 10, we demonstrate the ease of creating a robust client mashup through an example and evaluate the performance implications of realizing our abstractions. In Section 11, we compare and contrast with related work. In Section 12, we give a discussion on the future work and finally we conclude in Section 13.

2. BACKGROUND

The Web has evolved from a collection of static documents connected by hyperlinks into a dynamic, rich, interactive experience driven by client-side code and aggregation by Web services. The security policy of modern browsers was designed to avoid vulnerabilities in old sites, rather than to provide the best abstractions for the newest sites. In this section, we summarize the existing access control policies and the limitations they place on Web site design, then describe a new access control policy that mashup authors are demanding through existing proposals.

2.1 The Same-Origin Policy and the All-or-Nothing Trust Model

The *same-origin Policy* (SOP) governs the access control on today's browsers. The SOP prevents documents or scripts loaded from one origin from getting or setting properties of documents from a different origin [38]. (The origin that a script is loaded is the origin of the document that contains the script rather than the origin that hosts the script.) Two pages have the same origin if the protocol, port (if given), and host are the same for both pages. Each browser window, <frame>, or <iframe> is a separate document, and each document is associated with an origin. The SOP policy concerns three browser resources: cookies, the HTML document tree, and remote store access. In more detail, a site can only sets its own cookie and a cookie is sent to only the site that sets the cookie along with HTTP requests to that site. Two documents from different origins cannot access each other's HTML document using the Document Object Model (DOM) which is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents [13]. A script can access its document origin's remote data store using the *XMLHttpRequest* object, which issues an asynchronous HTTP request to the remote server [43]. (XMLHttpRequest is the cornerstone of the AJAX programming.) The SOP requires a script to issue XMLHttpRequest to only its document origin.

For example, an <iframe> sourced with http://a.com cannot access any HTML DOM elements from another <iframe> sourced with http://b.com and vice versa. *a.com*'s scripts can issue XMLHttpRequests to only *a.com*, but not to *b.com*. HTTP requests to *a.com* send only cookies that are set by *a.com*.

A document may contain <script> elements from different domains. Such third party scripts are treated as libraries that run as the document's origin rather than the scripts' origins and can access all of the document's resources. For example, a.com/service.html may contain the markup <script src='http://b.com/lib.js'>, which allows lib.js to access *a.com*'s HTML DOM objects, cookies and data through XMLHttpRequest. However, lib.js cannot access *b.com*'s resources since lib.js is associated with the site *a.com*, but not *b.com* in this context.

Following the SOP, today's browser abstractions offer an *all-or-nothing* trust model for Web programmers. Site *a.com* either does not trust Site *b.com*'s content at all by segregating *b.com*'s content into a frame or *a.com* trusts *b.com*'s scripts entirely by embedding *b.com*'s scripts and giving them full access to *a.com*'s resources.

2.2 Web Mashups

Web mashups are defined as Web sites that compose content from more than one site, yet this definition is in tension with the same-origin policy, which prevents such interactions. Many content providers want to publish information for any integrator site to use, but the same-origin policy prevents the integrator Web page from issuing XMLHttpRequests for the data directly.

Initially, mashup developers worked around these restrictions using a proxy approach: A proxy serves the aggregated content from various domains and appears to the browser to be "same-origin." The *pipes.yahoo.com* mashup creation wizard is a recent example of this approach. When a Web user visits the mashup on *pipes.yahoo.com*, it connects to *pipes.yahoo.com* to get data. The request is proxied to the real data provider, like NYTimes, and the response data is then passed back from *pipes.yahoo.com* to the

mashup. The drawbacks of this approach are that the content makes several unnecessary round trips, reducing performance; the proxy can become a choke point, limiting scalability; and the proxy can become a hop point that gives hackers more anonymity.

Recently, the *AJAX* (Asynchronous JavaScript And XML) programming model has emerged, allowing Web services to shift interactive user interface code from the Web server to the browser. Where conventional Web pages handle every click with a round-trip to the server, AJAX uses client-side code ("JavaScript") to handle many user interactions, providing interactivity not bounded by network and server performance. Furthermore, when communication with the server is required, that communication occurs asynchronously ("Asynchronous XML") through XMLHttpRequest, while the client-side code continues to provide interactivity in the meantime. Outlook Web Access and Google Maps are examples of early AJAX deployments.

The advent of AJAX makes *client-side* mashups popular. An early example is *housingmaps.com*, which mashes up the craigslist housing database with the AJAX Google Maps library. Client-side mashups include client-side interactions of script libraries from various sources as well as retrieving and aggregating data from different sites. The former requires fully trusting the incorporated, third-party scripts; the latter requires circumventing the same-origin policy and communication to servers from different domains. For the latter, by encoding public data in executable JavaScript format (JavaScript Object Notation, or JSON [30]), cross-domain <script> tags can also be used to pass data in executable format from the provider to the integrator across domain boundaries, eliminating the need for proxies. Using <script> tags for cross-domain data exchange has the unfortunate side effect of granting the integrator's privileges to the data provider, even though the data provider may not be trusted by the integrator. The all-or-nothing trust model of the SOP forces the integrator to trade security for the capability of cross-domain communications.

Web gadget aggregators, such as iGoogle [27] and Windows Live [33], are an advanced form of mashup, combining user-selected active content from third-party sources into a single portal page. A *gadget* is an HTML-plus-JavaScript component designed to be included into a gadget aggregator page; it is often the client side of some Web service. Gadget aggregators are security-conscious; they host each untrusted gadget in a frame on a distinct (sub)domain, relying on the SOP to isolate third-party gadgets from one another and from the outer page. However, because the SOP prevents interoperation among gadgets, aggregators also support *inline* gadgets, which include third-party code as a library of the aggregator page using the <script> tag. The all-or-nothing trust model of today's browsers unfortunately forces the gadget aggregator to decide between interoperation and isolation. Because inlining requires complete trust, Google's aggregator punts the security problem to the user: "Inline modules can...give its author access to information including your Google cookies...Click OK if you trust this module's author."

2.3 Verifiable-Origin Policy (VOP)

Frustrated by the limitations of the SOP, mashup developers have been pushing browser vendors to move to a new security policy for the Web, allowing fine-grained policy decision making between communicating domains. Web service providers want to be able to make their own access control decisions as to whether data they send between domains is public, whether it should be executed, or whether it should be ignored, rather than relying on the browser to make this decision. Several new browser communication proposals [11, 22] have emerged, governed not by the SOP, but rather by what we call the *verifiable-origin policy*: A site may request information from any other site, and the responder can check the origin of the request to decide how to respond. Communication that obeys the VOP is an important building block of our MashupOS proposal, and is discussed further in Section 7.

3. PRINCIPALS AND RESOURCES

3.1 Principals

We cast the world of Web applications in the context of the conventional notion of the multi-user operating system, in which different *principals* have access to different sets of *resources*. In the OS environment, the principal is a user or group. By associating a process with a principal, the OS ensures that the process only has as much power as the principal that controls the process' behavior. In general, one user does not trust another with respect to the confidentiality and integrity of her resources. In the Web environment, the principal is the owner of some Web content. With the same-origin policy, a principal on browsers is tied to the ownership of a DNS domain.

Other notions of principal have been explored. The cookie specification [16, 32] defines a path-based principal for cookie access. By default, a cookie is associated with the Web page that created it and any other Web pages in the same directory or any subdirectories of that directory. For example, if the Web page `http://www.example.com/catalog/index.html` creates a cookie, that cookie is also visible to `http://www.example.com/catalog/order.html` and `http://www.example.com/catalog/widgets/index.html`, but it is not visible to `http://www.example.com/about.html`. However, since today's SOP browsers allow same-domain pages to directly access one another's pages including their associated cookies, supporting such fine-grained, path-based principals is moot—scripts in `http://www.example.com/about.html` can (create then) navigate to an iframe sourced with `http://www.example.com/catalog/index.html` and access the index.html's cookie. It is possible for MashupOS to support a fine-grained principal and to reject legacy support for SOP principals. Then MashupOS must ensure that only "new" browsers and "new" servers interoperate. This saddles site operators with the problem of incremental deployment, providing both "MashupOS-enabled" content and legacy content. Once a site operator works around SOP principals for legacy compatibility, the proposed fine-grained principals offer no additional benefit. Therefore, we use the SOP principal as the MashupOS principal. Web servers that wish to provide more fine-grained principals must do so using DNS subdomains rather than subpaths. Hierarchical DNS names can represent hierarchical trust relationships among content owners. In many common Web deployments, unfortunately, content owners sometimes have no control of the server's DNS namespace. The rest of the paper uses the terms "domain" and "principal" interchangeably.

Because a browser is never used by more than one human user at the same time, MashupOS does not need to provide mechanisms for arbitrating browser resources among users. In contrast, content-owning principals can share browser resources simultaneously, and must be mediated by MashupOS.

3.2 Resources

Browsers provide to applications the following resources:

- *Memory*: The heap of script objects. This is analogous to process heap memory.

- *Persistent state*: Browsers provide applications with the ability to store a few kilobytes of cookies or some other persistent state [26], which persist across application invocations. This resource is weakly analogous to the OS file system.
- *Display*: The HTML DOM that controls the user's display, analogous to X Windows resources. The DOM API is itself a heap of JavaScript objects.
- *Network communications*: The ability to send and receive messages outside the application, equivalent to an OS network facility.

Later, we address how MashupOS assigns these resources to different domains, and to what extent domains are allowed to access resources belonging to other domains.

4. TRUST MODEL AMONG PRINCIPALS

An important goal of our work is to design abstractions that match common trust levels of Web programmers in their service creation, whether as a service provider providing a Web service or component or as a service integrator integrating or composing others' services into a mashup. The content-rendering capability and abstractions offered by browsers determine (and sometimes limit) how content is provisioned by service providers and how content can be integrated by service integrators from different domains. Today's browsers offer abstractions for only an all-or-nothing trust model, which is insufficient for today's Web services (Section 2). In this section, we analyze and derive the various content types in demand today as well as the common trust levels between providers and integrators, and we identify the new content types and the new abstractions that are needed for the missing trust relationships.

4.1 Existing Trust Relationship between Content Providers and Integrators

When a provider site *p.com* publishes an HTML file, say `http://p.com/p.html`, an integrator site *i.com* can integrate `p.html` only by using frames: For example, `http://i.com/i.html` contains <iframe src='http://p.com/p.html'>. By the same-origin policy (Section 2), `p.html` and `i.html` belong to different principals (domains), and hence cannot access each other's content on the browser. Here, the "access" is dictated by the DOM interface for content manipulation, which could be reading or writing a DOM or JavaScript object or function invocations. We call such HTML content *isolated content*. Providing isolated content means that the provider does not trust any integrators to access the provider's content. This isolation is mutual—the integrator cannot access the provider's content, and the content cannot access the integrator's resources including its HTML elements, cookies, and its remote store accessed through XMLHttpRequest. This trust relationship is indicated in Row 1 of Table 1.

When *p.com* publishes a script, say `http://p.com/p.js`, *i.com* can integrate `p.js` only by using the <`script`> tag: For example, `http://i.com/i.html` contains <script src='http://p.com/p.js'>. `i.com` treats `p.js` as library code, and `p.js` runs as the principal of *i.com* (rather than its source *p.com*). Because the same-origin policy applies only to the document origin, but not the script origin, a script from any domain can be included by a document of any domain even when their domains doesn't match, and the script's global variables, objects, or functions can be accessed by the document fully. For this reason, we call such script content *open content*. The open content can also access its integrator's resources fully.

Therefore, integrating open content requires the integrator to fully trust the provider's content to access any of the integrator's resources. It is possible that a provider includes in its open content private and sensitive data which the provider does not trust integrators to access directly. In such a scenario, the private and sensitive data is protected through server authentication or script data structures and scoping—the server only returns the script when the integrator is authenticated by the server; and the integrator can only access sensitive data inside the script through script global functions or methods. This trust relationship is shown in Row 3 of Table 1. However, this is a dangerous practice for providers. As exemplified by a recent Gmail vulnerability [21], attackers can lure a logged-in victim user to visit an attacker page which embeds the provider's script. Because the user is in an active session, the corresponding cookie that represents the user's credential is sent along to the provider for authentication of the script retrieval. The script that contains the sensitive data, the user's contact list in Gmail's case, is then returned to the attacker page. Although script data structure and scoping can perform some level of access control, it places high requirements on programmers' being careful and thorough.

Therefore, a provider should put only non-sensitive information in its open content. In that case, the provider can trust any integrator to access the content in an arbitrary way. This trust relationship is indicated in Row 5 of Table 1.

4.2 What is Missing

4.2.1 Cross-Domain Communications

What is lacking in the existing handling of isolated content is that providers have no way of offering controlled access to the isolated HTML content. This limitation has already motivated ad hoc ways of cross-domain communications, e.g., through fragment identifiers [6, 31] and common domain postfixes [28] as well as new proposals like JSONRequest [11] and cross-document messaging [22]. The latter proposals demand a VOP-based security policy that allows controlled access to providers' content based on the requester's source (Section 2). We call the isolated content to which the content owner provides access control through cross-domain communications, *access-controlled content*. The provider of the access-controlled content still does not trust any integrators to access the provider's content by default, but can use cross-domain communications to provide a subset of content based on the integrator's credentials. The integrators do not have to trust the provider and the provider cannot access the integrators' resources. The trust relationship for access-controlled content is shown in Row 2 of Table 1. We give our proposal of the *CommRequest* abstraction for cross-domain communications and the <`ServiceInstance`> abstraction for access-controlled content in Section 6.

4.2.2 Unauthorized Content

It is safe to have isolated or access-controlled content to run as its provider because no integrators are allowed unmediated access to the content. It also makes sense that open content does not run as the provider because integrators can directly access or tamper with the content, and instead runs as the integrator since the integrator is expected to trust the open content entirely. What is missing here is a type of content that an integrator can directly access, but does *not* trust to access the integrator's resources. For example, an integrator may want to use a script library (open content) from a different domain, but not trusting the script to touch the integrator's resources. It is necessary for such content to run as neither the provider nor the integrator. Here, we introduce *unauthorized content* to represent this kind of content. The trust relationship for unauthorized

	P trusts T to access P's content	T trusts P to access T's resources	Content type	Abstraction	Run-as Principal
1 2	No	No	isolated access-controlled	<Frame> <ServiceInstance> & *CommRequest*	Provider Provider
3	No	Yes	open	<Script> (bad practice)	Integrator
4	Yes	No	unauthorized	<Sandbox> <OpenSandbox>	None
5	Yes	Yes	open	<Script>	Integrator

Table 1: The Trust Model on the Web for a provider P and an integrator T

content is shown in Row 4 of Table 1. We present our abstractions <Sandbox> and <OpenSandbox> for unauthorized content and a provider-browser protocol for hosting and integrating unauthorized content in Section 5.

4.2.3 Support for Third Party Content

Another significant deficiency is a lack of support for hosted, third-party content which is a prevalent and fundamental part of today's Web. Examples of such content include user profiles of social networking Web sites like *MySpace.com*, user blogs at blog-hosting sites, and ads at ad-hosting sites.

The host uses different domains to partition and isolate content. Sometimes, a separate domain is used for each third party and for the host. For example, gadget aggregators iGoogle and live.com use separate domains for some of their third party gadgets. Again, this setup imposes complete cross-domain content isolation. As discussed above, a missing content type is *access-controlled content* that provides strong isolation across the principals while allowing the content owner to control others' access to its content.

Sometimes the host allows some third parties to share the same domain with the host. Domain-sharing allows content from different sources to be more tightly integrated, such as accessing one another's (DOM) objects by reference and directly invoking one another's functions. For example, iGoogle and live.com also support "inline gadgets", allowing third party gadgets to be inlined with the host page for interactions with the parent page as well as other gadgets. MySpace.com also hosts user profiles at the same domain. Unfortunately, modern browsers do not allow a host to differentiate its content from the content of third parties. That is, all content executes under the same credentials. This deficiency has already yielded dangerous consequences, such as Cross Site Scripting (XSS) attacks (Section 8). In one flavor of XSS attacks, malicious third party content, such as a user profile, is uploaded to the host and then abuses the host's privileges to cause damages to other third party content or the host itself. The notorious Samy worm [39] was such an attack that infected over a million MySpace.com users within just 20 hours, making Samy one of the fastest spreading worms of all time. What is missing is a protocol for the host to indicate the untrustworthiness of certain third party content to browsers and the capability for browsers to deny such content's access to any domain's resources by default unless explicitly allowed. Such content falls under *unauthorized content* whose objects and functions can be accessed directly by the host, but the third party script cannot access the host's resources unless explicitly allowed by the host.

Interestingly enough, different treatment of third party content is also demanded in the context of Web spam. Google announced in early 2005 [20] that hyperlinks with a `rel="nofollow"` attribute [23] would not influence the link target's PageRank [5]. In addition, the Yahoo and MSN search engines also respect this tag [44]. With a browser-server protocol and browser abstractions for unauthorized content in handling third party content, search engines can more easily fight against Web spam by discounting the links in unauthorized content.

4.3 Putting it Together: Completeness

Now we refer back to Table 1 to give a qualitative understanding of the completeness of our above derived content types and their associated browser abstractions demanded by the Web. Table 1 enumerates all possible trust levels between providers and integrators at the granularity of provider's content and integrator's principal. We don't strive to provide abstractions to satisfy all possible Web programmers' trust levels at the most fine-grained level, say the trust levels for a particular DOM element of an HTML content. Providing browser abstractions at such a fine-grained level would result in a large and complex set of abstractions that are hard to use correctly and difficult to reason about. We use different granularities for providers and integrators because when a provider provides a piece of content, the provider never intends to give away the rest of its principal's resources like cookies or access to remote store through XMLHttpRequest. However, when an integrator consumes content, all of the integrator's principal's resources are at stake.

As shown in Table 1, all trust levels are met with a content type along with either an existing abstraction (<Frame> or <Script>) or a new abstraction (<Sandbox>, <OpenSandbox>, <ServiceInstance>, *CommRequest*) which we will present in the subsequent sections.

For the security of its site, a provider must ensure that no matter how open content and unauthorized content may be used (or abused) by an integrator, it will not violate the access control of the provider's access-controlled or isolated content. For example, a provider that offers both an access-controlled mail service and a public map library service must ensure that its map library code or any other third party unauthorized content have no access to any of its users' mailboxes or contact lists.

5. UNAUTHORIZED CONTENT AND THE SANDBOX ABSTRACTIONS

In MashupOS, we enable service providers to publish and integrators to consume *unauthorized content* (Section 4), such as third-party content, without liability and overtrusting through a provider-browser protocol and the <Sandbox> and <OpenSandbox> browser abstractions, which we detail in this section. Allowing differentiation between third-party content and a host's own content has significant security benefits as we will discuss in Section 8.

5.1 Private and Open Unauthorized Content

A key property of unauthorized content is that such content is not authorized to *run as* any principal, and, hence, cannot have access to any principal's resources including its HTML DOM objects, cookies, and access to its remote store through XMLHttpRequest. Note that this is a one-way restriction that restricts the reach of unauthorized content, but integrators can possibly have full access to the unauthorized content by references.

Unauthorized content may be directly published by a service provider. This is the case for third party content hosting. An integrator can also turn an open content into an unauthorized content with our sandbox browser abstraction (described in detail in the next subsection) when the integrator does not trust the open content to access the integrator's resources.

Although unauthorized content does not run as any principal, it may *belong to* a principal in the sense that the unauthorized content is *private* to the principal and cannot be accessed by other principals. For example, an integrator may group some of its own, private HTML DOM elements (including its own scripts) and a third-party script (open content) into a piece of unauthorized content. In this scenario, the integrator would want to maintain this unauthorized content to be private to itself, and at the same time, disallow its access to the integrator's resources other than the private HTML DOM elements inside the unauthorized content. Private unauthorized content is particularly useful for an integrator to integrate open content that the integrator does not trust.

Open unauthorized content allows any principal to access the content directly. Open unauthorized content is particularly useful for providers to provide a service to any integrators while not being liable for it. For example, today's map services typically publish a script library and require integrators to supply a element for the map to be drawn. Alternatively, the map service could take the form of open unauthorized HTML content that already contains a element together with the map script library. This would save some steps for the integrators.

5.2 Sandboxing Unauthorized Content at the Browser

We introduce two new HTML tags for integrators to include unauthorized content: <Sandbox> for private unauthorized content that is hosted at and belongs to the integrator and <OpenSandbox> that may be hosted by any domain:

```
<Sandbox src='aFileName'>
   Fallback if tag not supported
</Sandbox>

<OpenSandbox src='aDomain.com/aFileName'>
   Fallback if tag not supported
</OpenSandbox>
```

Because the content in <Sandbox> is private, when the src attribute indicates a path from a different domain (principal), the enclosing page cannot access the content in the sandbox; only when the content comes from the same domain can the enclosing page access the content fully. In contrast, for <OpenSandbox>, no matter which domain hosts the content, the enclosing page can access the content fully including the HTML content. We use the term "sandbox" to loosely refer to either element.

Although the sandboxed content cannot reach out of the sandbox, the enclosing page of an open sandbox or the same-domain enclosing page of a private sandbox can access everything inside the sandbox by reference. The access includes reading or writing script global objects, invoking script functions, and modifying or creating DOM elements inside the sandbox through DOM method calls. When the enclosing page invokes a sandbox's function or method, the invocation is done in the context of the sandbox—it is analogous to calling setuid("unauthorized") before the invocation and setuid("enclosingPagePrincipal") after the invocation so that the enclosing page's principal's resources are inaccessible during the invocation. The enclosing page is unable to pass its own object references (or any other references that do not belong to the sandbox) into the sandbox. This is to prevent code from within the sandbox from following those references out of the sandbox. For example, the enclosing page is unable to pass its own display elements like into the sandbox. If an integrator wants to integrate a third-party library together with some of its own content such as display elements that may be needed by the library, the integrator should create its own (private or open) unauthorized content. The unauthorized content would include both the library and the display elements, which would all be sandboxed. In our implementation, every object reference from within the sandbox is checked to see whether the reference belongs to the sandbox; for references that don't belong to the sandbox, an exception is thrown.

5.2.1 Access control rules for sandboxes

The same-origin policy provides each piece of content with some freedom and some protection: It is free to access other content from the same domain, and it is protected from access by content from other domains. Sandboxes, however, are exceptions to these general rules. A private sandbox enjoys the usual protection, but an open sandbox enjoys less protection: It is intended to be used like a library to extend the enclosing piece of content (which may have a different origin); therefore an open sandbox can be accessed by exactly the same things that can access the enclosing content. A sandbox (whether open or private) has less freedom than other kinds of content: It is intended to hold untrusted code; therefore it can access only itself and by implication the nested open sandboxes that extend it, but not any other content even from the same domain, even if it's nested. If a sandbox were allowed to access nested same-domain content, malicious third-party code in the sandbox could gain access to private content by dynamically generating nested frames or private sandboxes. Since frames inside sandboxes are protected from tampering by the sandbox, they are not restricted by the sandbox and have all the freedom granted by the same-origin policy. Notice that the origin of a private sandbox is relevant for its protection but not for its freedom, and the origin of an open sandbox is completely irrelevant just like open content.

5.2.2 Sandboxes for robust client mashups

Most of today's client mashups use third-party scripts (open content) for service components. Sandboxes enable integrators to create robust and secure client mashups of open content without sacrificing the programming convenience of library function invocations. For each third-party library that an integrator uses, the integrator can generate a piece of (open or private) unauthorized content containing the library along with its needed display DOM elements such as and put the unauthorized content into a sandbox. The integrator can then access and mash up content across its sandboxes as it wishes without worrying about any of the libraries maliciously or recklessly tampering with the integrator's content or other resources. In Section 10.1, we demonstrate how easy it is to implement a more secure client-side mashup with <Sandbox>.

5.2.3 Risks

Although the code from within a sandbox can never follow ref-

erences to outside the sandbox, the data references from within a sandbox may be used by the outside of the sandbox. Since these data references are also managed by the untrusted library service, Web programmers should take caution and check the validity of these data before use.

5.3 Provider-Browser Protocol for Indicating and Rendering Unauthorized Content

When a provider publishes unauthorized content, a provider-browser protocol is needed for the provider to indicate to browsers that such content is not trustworthy and for browsers to forbid rendering (or running) the content in the name of any principal. Enforcement is needed not only for the new <Sandbox> and <OpenSandbox> tags that are designed to limit the reach of unauthorized content, but also for the existing browser abstractions like <frame>s — frames must not be allowed to render unauthorized content; otherwise, the unauthorized content would run in the name of the frame's principal and could access all of the principal's resources and violate the key semantic of unauthorized content.

We employ the MIME protocol [18] as our provider-browser protocol. We require providers of unauthorized content to prefix their MIME content subtype with `x-privateUnauthorized+` for private unauthorized content and with `x-openUnauthorized+` for open unauthorized content. For example, a private unauthorized HTML content must be labeled `text/x-privateUnauthorized+html`. We verified that both Firefox and Internet Explorer fail to render these MIME types.

To ensure that unauthorized content providers do not inadvertently publish unauthorized content as access-controlled/isolated content, we constrain the <Sandbox> tag to render only content that is of a private unauthorized MIME type and <OpenSandbox> to render only content that is of an open unauthorized MIME types or open content.

The integrator should take caution to sandbox open content consistently—if a third-party library is sandboxed in one application, but not sandboxed in another application of the same domain, then the library can escape the sandbox when both applications are used.

6. THE ABSTRACTION FOR ACCESS-CONTROLLED CONTENT

Existing mainstream browsers have no mechanism for controlled cross-domain communication. The only cross-domain communication primitive available is the <script> tag, which gives the service provider uncontrolled access to the integrator's domain. There do exist some cross-domain communication proposals (browser-to-server [1, 11, 22] and browser-side [41]), but they are not widely adopted, and they each exist in isolation, so none provides a general solution to the problem of mismatched trust patterns.

As a result of the lack of controlled cross-domain communication, there is also no way for a parent window and a child window, containing mutually untrusted content, to flexibly negotiate the layout of the boundary between them. Browser <frame>s offer isolation at the cost of rigid, parent-controlled layout; s offer flexible, content-sensitive layout at the cost of requiring full trust between parent and child content.

Finally, the only protection abstraction in contemporary browsers is the SOP boundary. Unlike an OS process, a single principal cannot instantiate this abstraction multiple times to provide fault containment among multiple applications.

In this section, we present the <ServiceInstance> abstraction, which is a unit of isolation, fault containment, and resource allocation. Controlled communication between <ServiceInstance>s is allowed through our *CommRequest* communication abstraction which we detail in the next section. The <ServiceInstance> abstraction is used for rendering access-controlled content (Section 4).

6.1 Isolation and Fault Containment: ServiceInstance

An application instantiates a <ServiceInstance> with the tag:

```
<ServiceInstance
  src="http://bob.com/app.html" id="bobApp">
```

The tag creates an isolated environment, analogous to an OS process, fetches into it the content from the specified `src`, and associates it with the domain *bob.com* that served that content. The HTML file specified by the `src` tag should contain only a `script` tag; any layout elements are ignored.

To understand the value of the <ServiceInstance> abstraction, we must specify how resources are *isolated*. A <ServiceInstance> is a unit of resource allocation; it accounts for commodity resources such as CPU and memory pages, as well as a protection boundary, to prevent other domains from compromising the privacy or integrity of the data stored in those resources. This paper defers the issue of commodity resource allocation to future work, and this section focuses on the concerns of protecting resources.

6.1.1 Memory

Each <ServiceInstance> has its own isolated region of memory: No <ServiceInstance> can follow a JavaScript object reference to an object inside another <ServiceInstance>. This is true even for <ServiceInstance>s associated with the same domain, just as multiple OS processes can belong to the same user: one domain can use <ServiceInstance>s to provide fault containment among multiple application instances. For example, suppose a browser runs both a calendar and an address book gadget from a single site. If the calendar application dies of an exception or runaway resource usage, its <ServiceInstance> can be killed without affecting the addressbook gadget. This fault containment is a bonus; it does not affect the adversarial cross-domain relationships that MashupOS focuses on.

6.1.2 Persistent state

Cookies are handled no differently than in existing browsers: two <ServiceInstance>s can access the same cookie data if and only if they belong to the same domain, just as two processes can access the same files if they are running as the same user.

6.1.3 Display

A raw <ServiceInstance> comes with no display resource. Instead, the parent document that created the <ServiceInstance> must allocate a subregion of its own display, called a <Friv> (more in the next section), and assign the <Friv> to the child <ServiceInstance>. A child can control multiple display regions if its parent assigns it multiple <Friv>s, just as a single process can control multiple windows in a desktop GUI framework, such as a document window, a palette, and a menu pop-up window. The code in the <ServiceInstance> controls each display region by manipulating its corresponding DOM tree. As Web applications grow in sophistication, and as sophisticated Web "window managers" appear to manage these applications [45], it will be important for MashupOS to support this pattern.

6.1.4 Network resources

A ServiceInstance can access its principal's remote data store through *XMLHttpRequest*, following the same-origin policy. ServiceInstances can also communicate with one another with *CommRequest* (Section 7.

6.2 Flexible Cross-Domain Display: Friv

In today's browsers, there are two primary mechanisms by which control of display regions can be parceled out to separate applications. The simplest is the <iframe>, which provides an entire logical browser window inside a rectangular frame in the parent page. If the enclosed page is bigger than its containing frame, scrollbars appear. Because the contained content is a separate page in a separate SOP domain, the child has no influence over the layout of its frame in the parent page. The alternative is the , which is the basic unit of layout. A can conform to the size of its contents. As a resizes, it affects the layout of its parent DOM element. Unlike the <iframe>, the is only a display layout interface, not a boundary between documents, so it provides no isolation. In practice, web developers often use s, sacrificing isolation to achieve flexible layout.

MashupOS introduces the <Friv>, a flexible cross-domain display abstraction. A <Friv>, like an <iframe>, provides a boundary between a container document and an inner document, isolating the content from separate domains, but enabling the inner document to appear within the container's display. Like a , the <Friv> allows the child's layout requirements to flow to the frame in the container, enabling the container to adjust its layout to suit the child document. It achieves this by providing default handlers that negotiate layout size across the isolation boundary using the MashupOS local communication primitives (Section 7), providing flexible -like layout behavior. The <Friv> is so named because it crosses the <iframe> and the .

The following tag syntax creates a new <Friv> in the parent's layout and assigns it to an existing <ServiceInstance>. The bobApp ServiceInstance receives an event referring to the initially-empty DOM tree for the <Friv>, and populates it using JavaScript code.

```
<Friv instance="bobApp">
```

This alternate syntax creates a new <ServiceInstance> and a new <Friv> simultaneously, and assigns the latter to the former.

```
<Friv src="http://bob.com/page.html">
```

6.2.1 ServiceInstance and Friv Life Cycle

By default, the life cycle of a <ServiceInstance> is limited by the <ServiceInstance>'s responsibility for some part of the browser's display. A <ServiceInstance> can track the display regions that it owns by registering a pair of handlers with the methods:

```
ServiceInstance.attachEvent('onFrivAttach', fn);
ServiceInstance.attachEvent('onFrivDetach', fn);
```

The first callback is invoked whenever the parent assigns a new <Friv> display to the <ServiceInstance>. When the parent reclaims the display associated with a <Friv> (by removing the <Friv> element from its DOM tree), the <Friv>'s DOM disappears from the child <ServiceInstance>'s object space, and the child's onFrivDetached handler is called.

The default onFrivAttached and onFrivDetached handlers track the set of <Friv>s. When the last <Friv> disappears, the <ServiceInstance> no longer has a presence on the display, so the default handler invokes

```
ServiceInstance.exit()
```

to destroy the <ServiceInstance>.

A <ServiceInstance> can act as a daemon by overriding the default handlers so that it continues to run even when it has no <Friv>s. Such a <ServiceInstance> may continue to communicate with remote servers and local client-side components, and has access to its persistent state. The browser may need to provide a process list, sorted by resource consumption, to facilitate killing errant daemons.

When a <Friv> is assigned to a new location (for example, using document.location = url, or equivalently, when the user clicks on a simple link in the <Friv>'s DOM), the <Friv>'s fate depends on the domain of the new location. The next two paragraphs describe the two possibilities.

If the domain is different from that of the <ServiceInstance> that presently owns the <Friv>, the behavior is just as if the parent had deleted the <Friv> (detaching it from the existing <ServiceInstance>) and created a new <Friv> and <ServiceInstance> with the <Friv src=...> tag. The only resource carried from the old domain to the new is the allocation of display real-estate assigned to the <Friv>. This behavior is analogous to creating a new process with a new identity, giving it the handle of the existing X Window region, and disconnecting the prior process from the same X Window.

If the domain matches that of the <ServiceInstance> that owns the <Friv>, then the HTML content at the new location simply replaces the <Friv>'s layout DOM tree, which remains attached to the existing <ServiceInstance>. Any scripts associated with the new content are executed in the context of the existing <ServiceInstance>.

Browsers allow a Web application to create a new "popup" window. The creation of a popup creates a new parentless <Friv> associated with the <ServiceInstance> that created the popup.

Given this definition of the life cycle of a <ServiceInstance> and <Friv>, the legacy <Frame> tag is implemented as follows: For each domain, there is a special "legacy" <ServiceInstance>. The <Frame src=*x*> tag is an alias for <Friv src=*x* instance=*legacy*>. Thus, all frame content and scripts for a single domain appear in a common object space, just as they do in legacy SOP-only browsers. Within the legacy <ServiceInstance>, each script still has a local document reference that identifies the <Friv> whose DOM the script was loaded with, so that references like document.location are meaningful.

Note that the automatic communication of layout information between a child document and its container introduces a path for information leakage. A malicious parent might load in a <Friv> the url //mail/search?q=enron, and infer from the layout size of the resulting child document how much email the user has regarding the search keyword. One defense would require documents to explicitly declare their willingness to participate in the <Friv> layout protocol, although MashupOS does not yet specify such a declaration.

7. COMMUNICATION: COMMREQUEST

As shown in Figure 1, a Web application is the pairing of browser-side components and server-side components, all owned by a common domain. This model guides the MashupOS design of communication among domains: to the extent that a browser-side application is an extension of a server-side application, it should be allowed to communicate with other domains in the same ways that the server can.

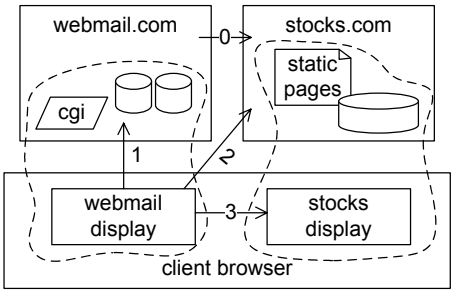

Figure 1: The Same-Origin Policy isolates HTML pages and frames based on their source. MashupOS supports both SOP (1) and VOP (1, 2, 3) communication patterns, mirroring those available to the server.

Legacy browsers follow the SOP in that they enable only communication from the browser-side component to its corresponding server (arrow 1). By communication we mean transfer of arbitrary data, such as an XML file. Servers, however, frequently communicate with other domains by establishing a new TCP connection (arrow 0). The recipient listens for the connection on a port, and with each connection learns the identity of the sender (its IP address, roughly equivalent to its SOP domain).

MashupOS follows the VOP, extending this facility to browser-side components by providing two additional communications paths: cross-domain browser-to-server communication and cross-domain browser-side communication.

7.1 Browser-to-server communication

Section 2 describes how the SOP protects legacy servers (such as those behind corporate firewalls) by confining browser-to-server communication to stay within the same SOP domain. We are not the first to observe [1, 11, 22] that cross-domain browser-to-server communication (arrow 2) can be safely allowed, so long as the protocol labels the request with the domain that initiated it, and any participating server verifies the domain initiating the request. A VOP-governed protocol must fail with legacy servers; we adopt the technique proposed by JSONRequest [11], requiring servers to indicate their compliance by tagging their replies with a special MIME content type (`application/jsonrequest`). Like JSONRequest, CommRequest transmissions omit cookies to avoid several subtle vulnerabilities.

7.2 Browser-side communication

MashupOS also provides browser-side communication across domains (arrow 3). For example, a <ServiceInstance> from *bob.com* may declare a `port` "inc", and register a handler function to receive browser-side messages on that port:

```
function incrementFunc(req) {
  var src = req.domain;
  return parseInt(req.body) + 1;
}
var svr = new CommServer();
svr.listenTo("inc", incrementFunc);
```

Another domain *alice.com* can address a browser-side message to Bob's port using the new URL scheme `local` that specifies Bob's SOP domain (⟨scheme, DNS host, TCP port⟩ tuple) and port name:

```
req = new CommRequest();
req.open("INVOKE", "local:http://bob.com//inc");
req.send(7);
y = parseInt(req.responseBody);
```

Local requests do not use HTTP, hence the special method INVOKE. This scheme of named ports forms a symbolic name service with nonconflicting names, since service names are required to include the SOP domain of the site sponsoring the service (the principal identifier).

Note that by requiring principal identifiers in the names, the *CommRequest* port naming scheme is not a general discovery service: an application cannot look up "local:IM" to discover any instant messenger service that happens to be running in the browser. We defer the task of a general discovery service (and the policy for resolving conflicts) to a service built over *CommRequest*.

That said, there may still be non-adversarial name conflicts from multiple <ServiceInstance>s running in the same browser. For example, if one service is instantiated twice, the second instance cannot listen to the same port as the first. In such an scenario, a developer may wish to address each instance by its DOM relationship to the message sender. To enable this relative routing, each <ServiceInstance> has a unique identifier (analogous to a process identifier), and a <ServiceInstance> can learn the identities of its parents and children.

For example, suppose both Alice's page and Bob's page include an instant-messaging gadget from *im.com*. Each parent page communicates with its own *im.com* <ServiceInstance> to set default parameters, or to negotiate <Friv> boundaries (Section 6.2).

The *im.com* <ServiceInstance> looks up its own identifier, and registers that identifier as a port name.

```
id = serviceInstance.getId()
svr.listenTo(id, imListenFunc);
```

Since the port's address includes the *im.com* principal, no other principal can maliciously introduce a port name conflict.

Alice's <ServiceInstance>, wishing to address its child *im.com* instance, does so using methods on the <ServiceInstance> element representing the child in the Alice's DOM:

```
si = document.getElementById("IMchild");
url = "local:"+si.childDomain()+si.getId();
```

Finally, a <ServiceInstance> can address its parent:

```
url = "local:"+serviceInstance.parentDomain()
  +serviceInstance.parentId();
```

The Web Applications working draft [41] proposes an alternative cross-domain browser-side communications mechanism. It provides only parent-to-child addressing, not global addressing between arbitrary browser-side components like the `local:` scheme in MashupOS. It does not yet specify data-only communications (although we surmise that is the intent). It reveals the full URI (not just the domain) of the sending document, which may reveal secret information such as session identification. It offers a unidirectional model; *CommRequest*'s asynchronous procedure call matches the *XMLHttpRequest* of today's deployed AJAX applications.

Our <Sandbox> and <OpenSandbox> are also allowed to communicate using *CommRequest* for both cross-domain browser-to-server messaging and browser-side messaging just in the same way as <ServiceInstance>. The origin of unauthorized content is labeled "unauthorized", and the protocol requires participating Web servers to authorize the requester before providing service.

Because the requester is anonymous, no participating server will provide any service that it would not otherwise provide publicly.

MashupOS restricts CommRequest messages to be data-only. As in JSONRequest [11], a *data-only object* is a raw data value, like an integer or string, or a dictionary or array of other data-only objects. Preventing the communication of arbitrary script objects discourages strong coupling between <ServiceInstance>s and from a sandbox to its enclosing page. Today it may be safe to pass a given object to another domain, but tomorrow another developer, modifying a different part of the application, may add a reference to the object, and suddenly the message recipient can reach arbitrarily deeply into the sender's object heap. Obviously, programmers have the power (eval()) to blow wide holes in their application's attack surface; this is no less true in the browser than when handling data-only server-to-server messages (Figure 1 case 0). The data-only restriction simply encourages designers to reason about their adversarial communication interfaces separately from their trusted internal object interfaces.

8. COMBATTING CROSS SITE SCRIPTING ATTACKS

As of 2006, more than 21% of vulnerabilities reported to CVE are Cross Site Scripting (XSS) vulnerabilities, ranking number one and surpassing buffer overflows, for two years in a row [7]. XSS is one of the consequences of insufficient protection and communication abstractions in today's browsers, particularly the lack of support for unauthorized content. In this section, we show how we use our abstractions ServiceInstance and sandboxes to combat XSS attacks in a fundamental way while retaining or even enhancing the richness of page presentation.

8.1 Background on XSS Vulnerabilities

In XSS, an attacker often exploits the case where a Web application injects user input into a dynamically generated page, without first filtering the input [29]. The injected content may be either *persistent* or *non-persistent*. As an example of a persistent injection attack, an attacker uploads a maliciously-crafted profile to a social networking Web site. The site injects the content into pages shown to others who view the profile. An injected script runs with the social networking site as its domain, enabling the script to make requests back to the site on behalf of the user. The notorious Samy [39] worm that plagued *myspace.com* exploited persistent injection; it infected over one million myspace.com user profiles within the first twenty hours of its release.

A malicious input may also be non-persistent, simply reflected through a Web server. For example, suppose a search site replies to a query x with a page that says "No results found for x." An attacker can trick a user into visiting a URL which contains a malicious script within the query x to the search site. The script in the reflected page from the search site will run with the search site's privilege.

8.2 Existing Defense

The root causes of the XSS attacks are unsanitized user input and unexpected script execution. Many existing mechanisms tackle the first cause by sanitizing user input. For applications that take text-only user input, the sanitization is as simple as enforcing the user input to be text, escaping special HTML tag symbols like < into their text form like "<". However, many Web applications, such as social networking Web sites like myspace.com, demand rich user input in the form of HTML. Because no existing browser abstractions constrain the reach of an included script, these Web sites typically have the policy of denying scripts in the user uploaded HTML pages. Consequently, user input sanitization involves script detection and removal. However, this turns out to be non-trivial: Because browsers speak such a rich, evolving language and many browser implementations exist, there are many ways of injecting a script [37]. In many occasions already, creative attackers have found new ways of injecting a script. The Samy worm [39] was notorious for discovering several holes in myspace.com's filtering mechanism.

The difficulty of exhaustive input filtering led researchers to tackle the second root cause, preventing unexpected script execution. Jim et al. [29] proposed BEEP to white-list known good scripts and adding a "noexecute" attribute to elements to disallow any script execution within that element. Eich also proposed a <jail> [14] that is similar to the "noexecute" . These proposals take a step towards the servers' goal of denying scripts. One drawback of these proposals is their insecure fallback mechanism when BEEP-capable or <jail>-containing pages run in legacy browsers: The "noexecute" attribute and the <jail> tag would be ignored by legacy browsers, allowing scripts in the and <jail> element to execute.

8.3 Using MashupOS for Defense

The reason behind Web servers' policy to disallow scripts is that existing browsers provide no way to restrict a script's behavior once it is included. The best known approach is to use a cross-domain <iframe> to isolate the user-supplied scripts. This approach is undesirable for three reasons:

- It requires the server to serve the scripts from a second domain (to associate the scripts with a distinct domain),
- The <iframe> provides an inflexible display layout, and
- The user-supplied content cannot interact, even in a constrained way, with its containing page.

MashupOS's <Sandbox>, <OpenSandbox>, and <ServiceInstance> solve all of the problems with <iframe> and can serve as a fundamental defense against XSS while allowing third-party script-containing rich content.

For persistent, user-supplied HTML content, a Web server can serve it as private or open unauthorized content (Section 5.3) depending on whether the content contains private data. For non-persistent user input in a reflected server page, the reflected page can use a sandbox or ServiceInstance to contain the user input with the "src" attribute being either a dynamic page proxied by the server:

```
<Sandbox
  src='userInput.asp?... escaped input ...'>
</Sandbox>
```

or a "data" URI [34] with encoded content:

```
<Sandbox
  src='data:text/x-privateUnauthorized+html,
       ... escaped user input ...'>
</Sandbox>
```

A <ServiceInstance> enables flexible layout by connecting the unauthorized content's display to the parent container with a Friv. In the case of sandbox, the unauthorized content's display DOM is directly accessible by the parent. A <ServiceInstance> can communicate with its parent's client or server components using the CommRequest primitive. A sandbox can do the same, plus the parent can directly access the child objects.

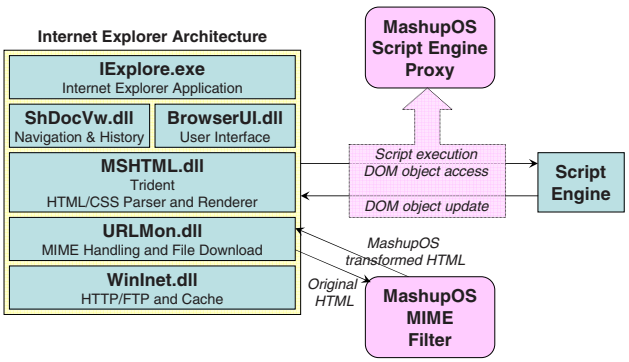

Figure 2: Our MashupOS prototype contains two extensions to Internet Explorer [25]: (1) MashupOS MIME filter that transforms HTML pages (2) MashupOS Script Engine Proxy that interposes DOM object access and update.

9. IMPLEMENTATION

We have built a MashupOS prototype system in which we have realized almost all proposed abstractions and their properties. We did not implement <OpenSandbox> and the enforcement of unauthorized MIME content types for sandbox abstractions. Also, our implementation supports only one Friv per ServiceInstance. Our prototype is based on Internet Explorer 7 (IE) and runs on both Windows XP SP2 and Windows Server 2003 SP1, but our methodology and techniques can also be applied to other browsers.

Instead of modifying IE's source code directly, we leverage browser extensions and public interfaces exported by IE. Figure 2 shows the MashupOS extensions. We don't anticipate that the actual adoption of our abstractions will be implemented with browser extensions, but inside browsers directly. The goal of our prototype implementation is to investigate how easily our abstractions can be realized.

Our system consists of two extensions to the IE architecture [25]. The first extension is a script engine proxy that we built from scratch using public interfaces exported by IE. As in all browsers, IE consists of an HTML/CSS rendering and layout engine and various script engines including a JavaScript engine and a VBScript engine. When a script element is encountered during HTML rendering, the script element is handed to a corresponding script engine for parsing and execution. Script execution may manipulate HTML DOM objects. For this purpose, the script engine asks the rendering page for references to needed DOM objects. We introduce the mechanism of *script engine proxy* (SEP), which interposes between the rendering engine and the script engines, and mediates and customizes DOM object interactions. To the rendering engine of a browser, a SEP serves as a script engine and exports the interface of a script engine; to the original script engine of the browser, the SEP serves as a rendering engine and exports the DOM interface of the rendering engine. We use *object wrappers* for the purpose of interposition. When a script engine asks for a DOM object from the rendering engine, a SEP intercepts the request, retrieves the corresponding DOM object, associates the DOM object with its wrapper object inside the SEP, and then passes the wrapper object back to the original script engine. From that point on, any invocation of the wrapper object methods from the original script engine goes through the SEP. In IE, a SEP takes the form of a COM [8] object and is registered in the Windows Registry associated with a scripting language (such as JScript) to serve as IE's script engine for that language. We have focused on the JavaScript language that is dominant in today's web applications. Our techniques can be readily applied to other languages.

Our MashupOS Script Engine Proxy (MSEP) takes the crucial role of implementing our various protection abstractions. Our general strategy here is to use the existing isolation mechanism, namely frames, as our building block. <ServiceInstance> and <Sandbox> are implemented using frames with at least one script inside (so that we can trigger MSEP's interposition on the frame). Isolation across protection domain boundaries is implemented with cross-domain frames while full access across protection domain boundaries is implemented with same-domain frames. For both kinds of sandboxes, we further mediate each object access from within a sandbox to ensure that the object belongs to the sandbox and not a reference from the outside of the sandbox. Our customized access control of cross-domain frames is realized via object wrappers as described above.

The second extension, MashupOS MIME filter, is an asynchronous pluggable protocol handler [35] at the software layer of URLMon.dll where various content (MIME) types are handled. Our MIME filter takes an input HTML stream and outputs a MashupOS-transformed HTML stream to the next software layer in IE. We use our MIME filter to translate new tags into existing tags, such as <iframe> and <script>; and we use special JavaScript comments inside an empty script element to indicate the original tags and attributes to MSEP. For example,

```
<sandbox src='unauthorized.uhtml'
         name='s1'>
</sandbox>
```

is translated by our MIME filter to

```
<script> <!-- /**
<sandbox src='unauthorized.uhtml'
         name='s1'>
 **/ --> </script>
<iframe src='unauthorized.uhtml'
        name='s1'> </iframe>
```

The comments inside the script element informs MSEP that the <iframe> with name "s1" should be treated as a sandbox. Similar translation happens to <ServiceInstance>.

Our CommRequest-based communication primitives are implemented by providing two runtime objects *CommServer* and *CommRequest*, with the communication methods described in Section 7. For access control on XMLHttpRequest and cookies, particularly for sandboxes, we again use the object wrapper mechanism above for interception.

<Friv> is implemented using <iframe> as well. We used CommRequest to carry out the automatic negotiation on the frame width and height between a <Friv> and its parent.

We find that script engine proxies can serve as a great platform for experimenting with new browser features. The fact that we are able to implement all our abstractions on this platform along with the MIME filter indicates that they should also be easy to add to the existing browsers.

10. EVALUATION

In this section, we first demonstrate the ease of programming robust Web services with MashupOS protection abstractions by

showcasing an example application in Section 10.1. We report the performance measurement of our prototype system in Section 10.2.

10.1 Showcase Application

We have implemented a photo location Web service, called PhotoLoc, as a showcase application. PhotoLoc mashes up Google's map service [19] and Flickr's geo-tagged photo gallery service [17] so that a user can map out the locations of photographs taken. Flickr provides API libraries in languages like Java or C# to interact with the Flickr Web server across the network, for example to retrieve geo-tagged photographs. To facilitate easy, browser-side cross-domain communication, we created an access-controlled, Flickr-based service that is isolated and protected with <Friv> (Section 6.2) and communications with the service are through *CommRequest* (Section 7). Google's map service is an open content script library (Section 4). PhotoLoc chooses an asymmetric trust relationship with Google's map library, which means that it trusts itself to use the library, but does not trust the library to access PhotoLoc's resources. PhotoLoc puts Google's map library along with the <Div> display element that the library needs into "g.uhtml" and serves "g.uhtml" as private unauthorized content. PhotoLoc's main service page (index.htm) uses <Sandbox> to contain "g.uhtml". PhotoLoc can access everything inside the sandbox, but Google's map library cannot reach out of the sandbox. Figure 3 shows our implementation.

10.2 Performance

Now we present the performance measurement of our MashupOS prototype. We first present our micro-benchmark results that measure the overhead of our MashupOS-enabled Script Engine Proxy (MSEP). Then, we present our macro-benchmark results on the impact of MSEP on page loading time. These measurements shed light on the performance implication of realizing MashupOS abstractions in the existing browsers. We conducted our measurements on a 1.7 GHz Pentium-4 PC with 1.5 GB of RAM, which runs Windows XP SP2 and Internet Explorer (IE) 7.

10.2.1 Microbenchmark on MSEP Overhead

We found a JavaScript and DHTML script performance benchmark called BenchJS [2]. The benchmark contains the following 7 JavaScript and DHTML tests.

1. Counting: count to 10000 and display a progress bar.
2. Open pops: open 8 pop ups and close them.
3. Replace images: replace 300 tiny images as fast as possible. It repeats this procedure 10 times.
4. Text manipulation: manipulate long text with different ways.
5. Set tables: create 2000 table-cells and calculates a random background color for each table-cell.
6. Put layers into place: create a phrase out of 50 different layers that are pulled together.
7. Calculate x-mas: calculate the days of the week for the next 10000 x-mas (not counting display time).

We run the 7 tests on both a MSEP-equipped IE 7 and IE 7 alone. We run each test four times and report the average latency in the top half of the Table 2. As we can see, for both the computational tests (Test 1, 4, and 7) that involve pure JavaScript objects and the tests with moderate interactions with DOM objects (Test 2 and 5), we observe negligible overhead. This is because these benchmark tests spend little time in MSEP's interposition logic. Test 3 and 6 involve heavy interactions with DOM objects and incur 82% and 33% overhead due to MSEP's DOM object interposition and manipulation.

Target	No-SEP	MSEP	Overhead
BenchJS benchmark			
1. Counting	1.56s	1.56s	0%
2. Open pops	8.09	8.09	0%
3. Replace images	1.18	2.15	82%
4. Text manipulation	1.47	1.50	2%
5. Set tables	2.89	2.95	2%
6. Put layers into place	5.22	6.93	33%
7. Calculate x-mas	4.94	4.98	1%
Our benchmark			
Pure JavaScript object	3.47μs	3.49μs	1%
DOM object	11.81	18.82	59%
Complex DOM operation	144.6	189.5	31%
Communication	17.15	14.05	-18%

Table 2: Microbenchmark results.

To further understand MSEP's overhead targeting its interposition functions, we designed our own set of micro-benchmarks as follows:

1. Pure JavaScript object: All 14 properties and methods of "Number" object.
2. DOM object: 92 properties and methods of "window", "document" and "clientInformation" object.
3. Complex DOM operation: 4 JavaScript statements including DOM traversal, element creation, style sheet update, inline script library loading, and event firing.
4. Communication: Cross-frame (same domain) function invocation when without MSEP; Cross-frame communication via CommRequest when with MSEP.

We measured the duration for 10,000 runs of each individual script statement with and without MSEP. We report the average time in the bottom half of Table 2. The results are consistent with that of BenchJS. MSEP incurs noticeable overhead for DOM manipulations, but negligible overhead for pure JavaScript object manipulations. Also, MashupOS's CommRequest is even more efficient than cross frame function calls.

10.2.2 Macrobenchmark Results

Our Macrobenchmark measurement evaluates the impact of MSEP on the overall page loading time. We picked the top 500 pages from the top click-through search results of MSN search from 2005. In our measurement, we disabled the browser cache, measured the page loading with and without MSEP. We implemented a Browser Helper Object [4] to capture the time before a document navigation and after the document loading. From our

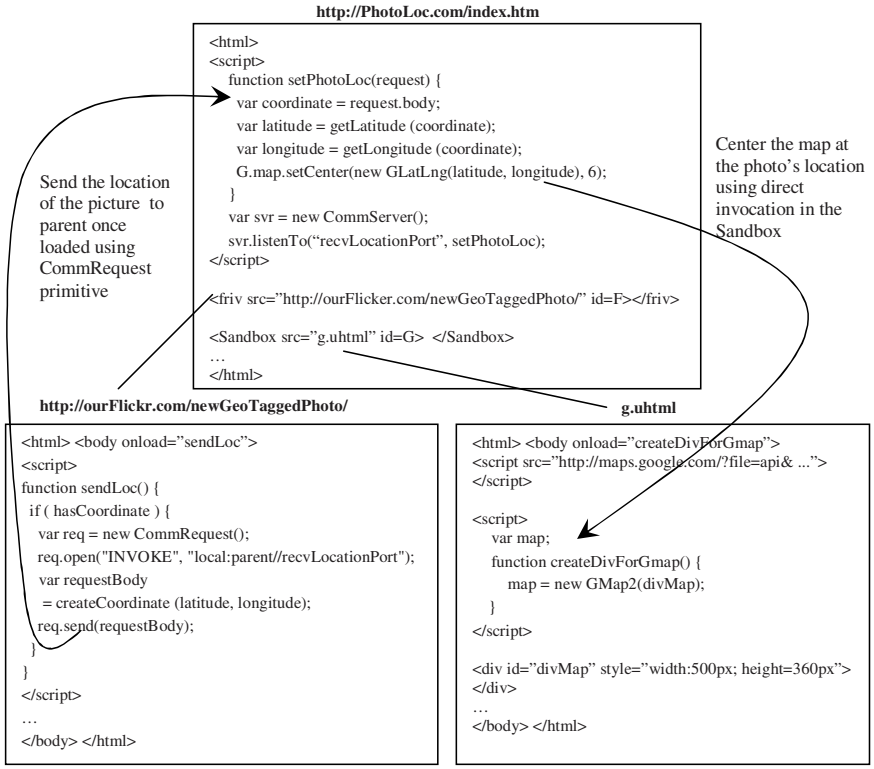

Figure 3: Our showcase PhotoLoc Service using `Friv`, *CommRequest*, and `<Sandbox>`

numbers, we do not observe an MSEP impact on the page loading time. This is because script execution time is short, on the order of microseconds, compared to the time for page retrieval over the network and the page rendering, which is on the order of seconds. Among the 500 pages that we measured, 30 of them cannot be loaded or rendered correctly in IE 7, and MSEP caused just one more faulty page than that. This gives us confidence on the completeness and robustness of our prototype.

11. RELATED WORK

11.1 Web Desktops

Recently, a new wave of "Web operating systems" [15] or web desktops [42], such as YouOS [45], have emerged. These sites present a traditional desktop user interface, complete with a window manager. However, all the applications are JavaScript code hosted from the same domain. There is actually no operating system that performs resource protection and management.

11.2 Existing Proposals for Isolation and Communications in Browsers

Crockford proposed a new HTML `<module>` tag to partition a page into a collection of modules [12]. A module groups DOM elements and scripts into an isolated environment; VOP-based socket communications can be used to transmit data in JSON format between the inner module and the outer module. To isolate the module from the origin server, modules may not make network requests. Thus, modules can be realized by our private sandbox enclosed in a ServiceInstance.

We have given detailed explanation and comparison with existing cross-domain communication proposals such as JSONRequest [11] and XDM [22] in Section 7. The Flash browser plugin offers a less fine-grained form of communication using cross-domain policy files [1]. A service provider can place a crossdomain.xml file on the server, and Flash uses the contents of this file to determine which service integrator domains can access files on the provider's server. Free libraries are available to allow JavaScript to access these Flash communication primitives [9]. Although this approach provides more flexibility and control than standard SOP communication model, the provider's server cannot distinguish between network requests originating from its own code and requests generated by integrators that are permitted by the crossdomain.xml file.

Subspace [28] provides a cross-domain communication mechanism in the context of gadget aggregators and is designed to run on current browsers without any additional plug-ins or client-side changes. Subspace splits a site into subdomains using each subdomain for a gadget (a principal). A subdomain can in turn be used to draw in scripts from other domains. Cross-subdomain communication channel is set up by setting `document.domain` of two subdomains to a common domain postfix. Subspace requires significant work on the part of the Web developer to use correctly, particularly for complex mashups with untrusted code from many different sources.

Fragment identifiers, the string after the # in a URI, have also been used for cross-domain, cross-frame communications [6, 31] because `window.location` can be modified (though not read) by a cross-domain frame and modifying fragment identifier does not cause document reload. This scheme requires careful synchronization between the communicating pages, and can be easily disrupted if the user presses the browser's back button. These schemes are temporary rather than long term solutions to cross-domain commu-

nications. Browsers should provide built-in cross-domain communication primitives.

11.2.1 Sandboxing

"Sandbox" is an overloaded term. Oftentimes, it broadly means an isolated environment.

Our sandbox abstraction is named this way because it shares many properties with the original sandbox proposal by Wahbe et al. [40]. Wahbe et al.'s sandbox addresses the scenario where desktop applications contain (untrusted) third-party libraries or modules. They use software-based fault isolation (SFI), namely, binary rewriting to contain the reach of untrusted modules into a fault domain (a segregated and contiguous region of memory) in the same address space as the application's process. The mechanism also manifests an asymmetric access pattern: The application's process can access the fault domains inside its address space, but the fault domains cannot reach out. However, the primary motivation for this sandboxing work was to avoid the high overhead of context switches if the untrusted code were placed in a separate process, rather than to have an abstraction for realizing an asymmetric access pattern that is needed for integrating unauthorized content or untrusted open content. In fact, people have not paid much attention to or leveraged the property of asymmetric access in the original sandboxing. In MashupOS, we adapt sandboxing to the browser environment to particularly leverage this asymmetry to match a common trust scenario between providers and integrators and provide both security and ease in creating client mashups.

Cox et al. [10] also recognized that browsers have become a de facto operating system for executing client-side components of Web applications but offer insufficient isolation across Web applications and can cause the desktop machine to be infected by drive-by-downloads or by attacks that exploit browser vulnerabilities. To this end, Cox et al. proposed the Tahoma browser operating system as a system layer beneath the running browser instances. Tahoma puts each "Web application" into a separate virtual machine. The "Web application" consists of a browser instance and a set of Web sites that the browser instance is permitted to visit. The use of the virtual machine is for isolation among the "Web applications" as well as between a browser and its host system. However, Tahoma does not address more fine-grained protection and communication among the sites within a "Web application" or a site. In comparison, MashupOS aims to enhance the browser itself to be a multi-principal OS that mediates resource access from different sites, and does not defend against attacks exploiting browser vulnerabilities or infections through drive-by download. MashupOS-enabled browser plus the Tahoma system layer would provide both fine-grained isolation and host system protection.

12. FUTURE WORK AND DISCUSSIONS

Today's browser extensions, including ActiveX controls and browser plugins can provide abstractions to Web pages that enable arbitrary flows of information within the browser and across the network. Unfortunately, these extensions may weaken the protection provided by MashupOS. Tools can be developed to identify the extensions through which browser security policies can be circumvented and users should be warned before such extensions are installed.

All major browsers have had cross-domain vulnerabilities that can be exploited to circumvent the SOP. It is an open challenge to have a robust browser implementation that guarantees the isolation boundaries of the browser security policy to be obeyed. Reis et al. realizes the SOP isolation using OS processes [36], putting each domain into a separate OS process for isolation. A number of operating systems [3, 24] have been built using safe languages to enable efficient and robust implementation of protection domains. These may be promising directions for robust implementation of browsers.

We have only explored the protection and communication issues in this paper. Many other issues, such as resource management and useful OS facilities that browsers can offer to Web services, deserve further study.

As discussed in Section 4, Web servers must segregate their open, isolated/access-controlled, and unauthorized content strictly so that no one can abuse the open or unauthorized content to access the isolated/access-controlled content. Tools are needed for Web servers to check and ensure the segregation of these content types. Information flow-based techniques may be applicable here.

13. CONCLUDING REMARKS

The advent of AJAX and client mashups have turned Web browsers into a multi-principal operating environment. However, browser support for Web programmers has lagged behind and remained in a single-principal world. The MashupOS project is building a multi-principal operating system for Web browsers. This paper focuses on the most imminent needs of today's browsers: abstractions for protection and communication.

From our analysis over the trust relationship between content providers and integrators, we derived four content types that require support from the Web and browsers: isolated content, access-controlled content, open content, and unauthorized content. Existing browser abstractions support only the isolated content with cross-domain frames and open content with scripts, resulting in an all-or-nothing trust model—an integrator site either trusts a provider site entirely by including the provider's scripts or does not trust a provider at all by putting the provider content inside a frame. Inflexibility and insecurity result from the incomplete support of today's browsers.

In MashupOS, we have proposed abstractions for the missing content types and trust relationships. We introduce *unauthorized content* as a fundamental addition to today's Web content provisioning. Our sandbox abstractions and provider-browser protocol enable content providers to publish and integrators to consume unauthorized content without liability and overtrusting, providing both security and ease in creating client mashups. Such support can fundamentally combat Cross Site Scripting attacks (a prominent threat in today's Web) while allowing the richest possible third party content. We have also proposed *ServiceInstance* for isolation, fault containment, and as the unit of resource allocation and *CommRequest* as a VOP-based communication abstraction unifying existing cross-domain communication proposals.

Our abstractions are backward compatible, allowing Web programmers to supply alternative content for browsers that don't support our abstractions. We have carefully designed the MashupOS abstractions to avoid unintended interactions between new content that use our abstractions and legacy ones. These enable Web programmers to adopt our abstractions with ease and comfort.

Our MashupOS prototype realizes almost all our proposed abstractions and their properties. Our evaluation showcases an easy-to-build and robust client mashup. Measurement of our prototype shows negligible overhead. The implementation and evaluation demonstrate the ease of adding these abstractions to existing browsers.

14. ACKNOWLEDGEMENTS

We'd like to thank Andy Begel, Shuo Chen, Adam Costello,

Douglas Crockford, Richard Draves, John Dunagan, Sunava Dutta, Hank Levy, Charlie Kaufman, Jay Lorch, Charlie Reis, Yinglian Xie, Zhenbin Xu, and anonymous reviewers for their valuable discussions and feedback to our work and this paper.

15. REFERENCES

[1] Adobe. External data not accessible outside a Macromedia Flash movie's domain, 2007. http://www.adobe.com/cfusion/knowledgebase/index.cfm?id=tn_14213.

[2] JavaScript Speed Test: BenchJS. http://www.24fun.com/downloadcenter/benchjs/benchjs.html.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, December 1995.

[4] Browser Helper Object. http://msdn2.microsoft.com/en-us/bb250436.aspx.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *7th International World World Web Conference*, 1998.

[6] J. Burke. Cross Domain Frame Communication with Fragment Identifiers. http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html.

[7] S. M. Christey. Vulnerability Type Distribution in CVE, September 2006. http://www.attrition.org/pipermail/vim/2006-September/001032.html.

[8] Component Object Model (COM). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/componentobjectmodelanchor.asp.

[9] J. Couvreur. FlashXMLHttpRequest: cross-domain requests. http://blog.monstuff.com/FlashXMLHttpRequest.

[10] R. Cox, J. Hansen, S. Gribble, and H. Levy. A Safety-Oriented Platform for Web Applications. In *Proc. IEEE Symposium on Security and Privacy*, 2006.

[11] D. Crockford. JSONRequest. http://www.json.org/jsonrequest.html.

[12] D. Crockford. The Module Tag: A Proposed Solution to the Mashup Security Problem. http://www.json.org/module.html.

[13] Document Object Model. http://www.w3.org/DOM/.

[14] B. Eich. JavaScript: Mobility and Ubiquity. http://kathrin.dagstuhl.de/files/Materials/07/07091/07091.EichBrendan.Slides.pdf.

[15] Big WebOS roundup - 10 online operating systems reviewed. http://franticindustries.com/blog/2006/12/21/.

[16] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, August 2006.

[17] Flickr Services API. http://www.flickr.com/services/api/.

[18] N. Freed. *Media Type Specifications and Registration Procedures*, December 2005. http://rfc.net/rfc4288.html.

[19] Google Maps API, 2007. http://www.google.com/apis/maps/.

[20] Preventing comment spam, January 2005. http://googleblog.blogspot.com/2005/01/preventing-comment-spam.html.

[21] J. Grossman. Advanced Web Attack Techniques using GMail. http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html.

[22] W. H. A. T. W. Group. Web Applications 1.0, February 2007. http://www.whatwg.org/specs/web-apps/current-work/.

[23] HTML 4.01 Specification, December 1999. http://www.w3.org/TR/html401/.

[24] G. Hunt and J. Larus. Singularity: Rethinking the Software Stack. In *Operating Systems Review*, April 2007.

[25] Internet Explorer Architecture. http://msdn.microsoft.com/workshop/browser/overview/ie_arch.asp.

[26] Persistence of Internet Explorer. http://msdn.microsoft.com/workshop/author/persistence/overview.asp?frame=true.

[27] G. Inc. Google Gadgets API Developer Guide. http://www.google.com/apis/gadgets/fundamentals.html.

[28] C. Jackson and H. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *Proc. WWW*, 2007.

[29] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-Enforced Embedded Policies. In *16th International World World Web Conference*, May 2007.

[30] JavaScript Object Notation (JSON). http://www.json.org/.

[31] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and s. Yoshihama. SMash: Secure Cross-Domain Mashups on Unmodified Browsers. Technical report, IBM Research, Tokyo Research Laboratory, June 2007.

[32] D. Kristol and L. Montulli. HTTP State Management Mechanism. IETF RFC 2965, October 2000.

[33] Windows Live Gadget Developer's Guide. http://microsoftgadgets.com/livesdk/docs/default.htm.

[34] L. Masinter. RFC 2397: The "data" URL Scheme, August 1998. http://tools.ietf.org/html/rfc2397.

[35] About Asynchronous Pluggable Protocols. http://msdn2.microsoft.com/en-us/library/aa767916.aspx.

[36] C. Reis, B. Bershad, S. Gribble, and H. Levy. Using processes to improve the reliability of browser-based applications. In *Under submission*.

[37] RSnake. XSS Cheat Sheet. http://ha.ckers.org/xss.html.

[38] J. Ruderman. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html.

[39] Technical explanation of The MySpace Worm. http://namb.la/popular/tech.html.

[40] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[41] Web applications working draft. http://www.whatwg.org/specs/web-apps/current-work/#crossDocumentMessages.

[42] Web desktop. http://en.wikipedia.org/wiki/Webtop.

[43] The XMLHttpRequest Object. http://www.w3.org/TR/XMLHttpRequest/.

[44] Google, Yahoo, MSN Unite On Support For Nofollow Attribute For Links, January 2005. http://blog.searchenginewatch.com/blog/050118-204728.

[45] YouOS. http://www.youos.com/.