# Subspace: Secure Cross-Domain Communication for Web Mashups

Collin Jackson
Stanford University
collinj@cs.stanford.edu

Helen J. Wang
Microsoft Research
helenw@microsoft.com

## ABSTRACT

Combining data and code from third-party sources has enabled a new wave of *web mashups* that add creativity and functionality to web applications. However, browsers are poorly designed to pass data between domains, often forcing web developers to abandon security in the name of functionality. To address this deficiency, we developed *Subspace*, a cross-domain communication mechanism that allows efficient communication across domains without sacrificing security. Our prototype requires only a small JavaScript library, and works across all major browsers. We believe Subspace can serve as a new secure communication primitive for web mashups.

## Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized Access*

## General Terms

Design, Security, Performance

## Keywords

access control, trust, web services, same origin policy


**Figure 1: Example warning message from Google personalized homepage. This third-party "todo" list gadget requires inlining so that it can request additional height from its parent as more items are added to the list. Unfortunately, inlining also allows the gadget author to access the user's Google account.**

## 1. INTRODUCTION

A mashup is a website or web application that seamlessly combines content from more than one source into an integrated experience. Recently these websites have been on the rise. For example, www.housingmaps.com combines Google Maps data with Craigslist's housing data and presents an integrated view of the prices of the houses at various locations on the Google map. Gadget aggregators, such as Microsoft Windows Live and Google Personalized Homepage, aggregate third-party JavaScript code, the *gadgets*, into one page to provide a desirable, single-stop information presentation to their users.

Because mashups bring together content from multiple sources, they must somehow circumvent the traditional *same-origin* web security model to obtain third-party data. Often web developers are forced to chose between security and functionality. Current practices include giving uncontrolled cross domain execution through the use of <script> tags and extending the browser with plugins for cross domain interactions. The former incurs high security risks because one site gets complete control over the other, while the latter is inconvenient for users whose browsers are not supported by the plugin or who are unwilling to install new software.

Efficient cross-domain communication is particularly important fo gadget aggregators, such as Google Personalized Homepage and Microsoft Windows Live. These gadget aggregators typically are presented with only two security choices: run gadgets *inline* in the same domain as the gadget aggregator, or *sandbox* them in frames with different domains to ensure that they cannot read or write the aggregator page. An example "todo list" gadget [1] that requires inlining is shown in Figure 1. Sandboxed cross-domain frames cannot engage in client-side communication with the parent frame, so this gadget must be inlined to communicate efficiently with its parent.

New proposals for cross-domain communication mechanisms [3, 6, 12] could deliver these much-needed cross-domain communication features directly into browsers, but these technologies are still years away and cannot be relied upon by websites until available in all the most common browsers.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
*WWW 2007*, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

In this paper, we present Subspace,[1] a communication primitive that enables cross-domain network requests and client-side communication, while protecting the aggregator from malicious web services or gadgets. Our mechanism is practical as it combines existing browser features and requires no client-side changes. Like existing gadget aggregators, we sandbox gadgets or web services using cross-domain frames. We then enable cross-domain communication across frames by setting up a *Subspace communication channel* between the aggregator and each gadget. To establish the channel, we pass JavaScript objects across the frames by manipulating the document domain property of the frames. Peer cross-domain gadget communication is mediated by the aggregator and carried out using these channels.

We have developed prototype mashups using Subspace that work on all major browsers (Internet Explorer 6, Internet Explorer 7, Mozilla Firefox, Safari, and Opera). Our timing measurements show that Subspace takes longer for the initial setup than existing practices, but provides fast and safe communication once the Subspace communication channels are in place.

Section 2 surveys existing practices for cross-domain communication, and Section 3 describes the key browser features that Subspace relies on. Section 4 describes the process for cross-domain communication with a single untrusted web service, while Section 5 extends this technique for multiple untrusted web services. Results of our timing measurements are presented in Section 6. We provide further discussion in Section 7 and survey related work in Section 8.

## 2. CURRENT PRACTICE

In this section, we explain how mashups currently communicate across domains. We begin by describing the same origin policies that are designed to prevent such communication.

### 2.1 Same-origin policies

Web browsers allow users to view interactive content without completely trusting the owner of that content. An untrusted web page cannot observe or corrupt the user's actions at other sites, nor can it issue unwanted transactions on behalf of the user. This sandbox model is designed around the idea of the same-origin principle, which states that "only the site that stores some information in the browser may read or modify that information." [8]

Because the web relies on interconnections between sites, this principle is not interpreted literally but rather applied loosely as a collection of *same-origin policies* on individual browser features, such as cookies, JavaScript access to documents, and plugins (Flash, Adobe Reader, and Java). These policies are designed to allow some exceptions that are considered beneficial, voluntary cooperation between sites.

Although distinct same-origin policies exist for many web features such as cookies, the restrictions that are of greatest interest to web mashups are the JavaScript restrictions that regulate access to inline frames (`IFRAME`s) and the XMLHttpRequest object. Inline frames can be used to download rich HTML documents from outside sources, but if the content comes from a different domain, the browser will not allow the JavaScript in the containing page to read or manipulate the document inside the frame, and vice versa. The XMLHttpRequest can be used to download arbitrary XML documents without rendering them in a browser pane, but it cannot be used to download files that are not from the same domain as the page making the request.

By enforcing these restrictions, the JavaScript same-origin policy protects the secrecy of HTML documents that the user has access to, and also protects the integrity of a page against unauthorized modifications by other pages.

[1]We borrowed this name from Star Trek, where "subspace communications" are used to establish instantaneous contact with people and places that are light-years away.

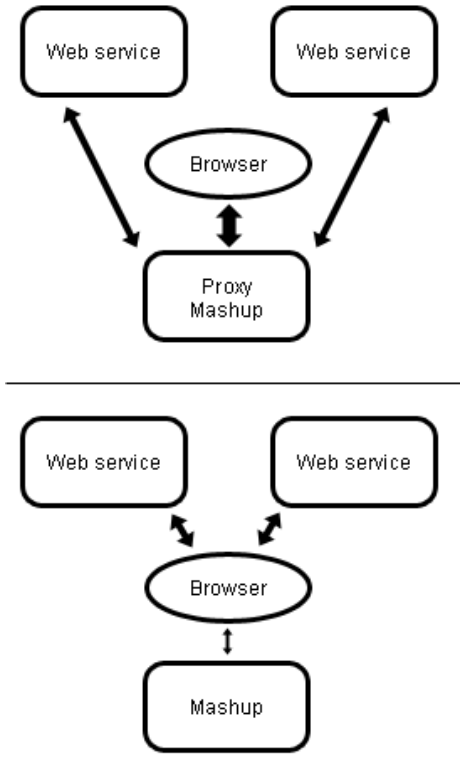


**Figure 2: Mashups can reduce bandwidth and latency by switching from a proxy approach (top) to a unsafe cross-domain `<script>` tags (bottom). Subspace provides a safer alternative to cross-domain `<script>` tags.**

### 2.2 Proxies

The website hosting the mashup can host a URL which relays data between the client and the source of the data. These proxies (sometimes known as *bridges*) make the data appear to the client to be "same-origin" data, so the browser allows this data to be read back out of an `IFRAME`, or more commonly, an XMLHttpRequest.

As shown in Figure 2, a disadvantage of this approach is that it adds the latency of connecting through mashup's proxy server, which generally takes longer than connecting directly to the server hosting the data. Bandwidth costs for the mashup author are also increased by the proxy approach, particularly if the size of the mashup's code is small relative to the amount of cross-domain data being proxied.

Although these proxies only go to one place, because they are left open for anyone to use, they provide another layer for hackers to hide behind for denial of service or exploiting input validation vulnerabilities on the server hosting the data source.

## 2.3 Cross-domain <script> tags

The same-origin policy on JavaScript protects HTML documents loaded from other domains, but it does not apply to scripts, which can be loaded from other domains and executed with the privileges of the page that included them. These remote scripts can be added to the page dynamically, allowing new data to be loaded into part of a page without forcing the entire page to be loaded. Unlike an XMLHttpRequest, which provides full read access to the content being requested, a script can only be accessed by executing it. This restriction is important because it ensures that only valid JavaScript files can be accessed across domain boundaries; all other files such as HTML documents will cause a syntax error.

Execute-only access requires that the page including the script fully trusts the source of the script. The page including the script has no way of performing input validation to ensure that the script being retrieved is not misusing its access to the parent page. The site hosting the script could change the content of the script at any time, and could even serve different scripts to different users.

Other cross-domain tags that can transmit information include stylesheets, which have the same security issues as scripts, and images, which can transmit limited information through height and width.

## 2.4 Browser plugins

Macromedia's Flash browser plugin includes its own scripting language based on JavaScript, called ActionScript. Macromedia's Flash plugin can support communications between domains via the use of a special crossdomain.xml file. This file is placed on the server that wishes to open up some or all of its files to cross-domain reading. Before allowing a cross-domain data request with ActionScript, Flash checks to ensure that the file exists and that the policy allows access. For example, the following file allows open access to files in its directory and all subdirectories:

```
<cross-domain-policy>
    <allow-access-from domain="*"/>
</cross-domain-policy>
```

Although browser plugins can provide many of the cross-domain network communication capabilities that are needed by mashups, some users choose not to install them for security, privacy, or compatibility reasons.

## 2.5 Fragment identifier messaging

Each browser has certain objects in the browser that can be accessed despite that object belonging to another domain. [9] One important example is the `window.location` object, which can be set (but not read) by frames of another origin. If a frame from Site A can access a frame of a page from Site B, it can pass a message to Site B by setting the location of Site B's page to be equal to Site B's current location plus a fragment identifier, starting with #. Because browsers do not reload the page when navigating to a fragment, the Site B page is not interrupted, but can receive the message without any network requests being sent. This technique is known as Fragment Identifier Messaging and has been used by some mashups to pass information on the client side between frames. Unfortunately, it requires careful synchronization between the communicating pages, and can be easily disrupted if the user presses the browser's back button.

# 3. BUILDING BLOCKS OF SUBSPACE

In this section, we provide background information on the building blocks of Subspace.

## 3.1 Cross-subdomain communication

One fuzzy aspect of the same-origin principle is the notion of a "site." For purposes of JavaScript security, a site is defined as the triple (protocol, hostname, port) [11]. For example, http://a.example.com and http://b.example.com are considered to be different sites, while http://www.example.com/a and http://www.example.com/b are considered to be the same site.

However, if two domains that want to communicate share a common suffix, they can use the JavaScript `document.domain` property to give each other full access to one another. This variable defaults to the host name of the server that the document was retrieved from, but can be changed to a suffix (and only a suffix) of this name. For example, pages on `a.example.com` and `b.example.com` can change the value of `document.domain` to `example.com`, allowing them to pass JavaScript data and code between each other at runtime.

Once a page has changed its domain using this mechanism, it is no longer permitted to access other frames that do not match its new domain. Further changes to document.domain can only shorten it, not lengthen it. Changing document.domain to top level domain names (e.g. "com") is not allowed, preventing this technique from being used for communication with arbitrary domains. However, the technique described in Section 4 allows sites to work around this restriction.

## 3.2 Cross-domain code authorization

Normally, the same-origin policy prevents code from passing between domains. A function defined in one domain will not be called by code in another domain, so there is no ambiguity about which domain is performing an action when the same-origin security checks are applied. However, using the data-passing technique described in Section 3.1, functions can be passed between domains.

A closure is a function that refers to free variables in its lexical context. It is associated with an environment that binds those variables. Typically, a closure is defined within the body of another function, referencing variables that were in scope when it was created, but are not in scope when it is called [2].

An example of a closure is this function, which returns the height of the current document in the user's web browser:

```
function h() { return document.body.clientHeight; }
```

This closure would provide a useful service if provided by a child frame to its parent, because it would allow the parent frame to find out the height of the child frame, ensuring that all of its contents are visible and no scrolling is necessary.

By setting the `document.domain` variable, a web page could pass a closure to a frame in another domain. The ECMAScript specification [7] does not provide an authorization policy for such closures, so browser vendors have arbitrarily picked their own behavior for this situation. There are two reasonable solutions, each of which has some adoption:

- **Dynamic authorization.** The closure inherits the security privileges of the page that is calling it. This approach corresponds to following the control links of the call stack, in a manner similar to stack inspection [5]. For example, if Site A calls a closure that was obtained from Site B, the closure would be able to access any browser state (DOM objects, cookies) associated with Site A, but not Site B. Our testing has determined that this approach is adopted in Opera and Safari.
- **Static authorization.** The closure inherits the security permissions of the page where the closure was created. This approach can be implemented by following the function's access link instead of the control link. For example, if Site A calls a closure that was obtained from Site B, the closure would be able to access any browser state associated with Site B, but not Site A. Our testing has determined that this approach is adopted in Internet Explorer and Firefox.

Static authorization allows for greater flexibility in delegation and asynchronous event-driven communication. It allows one domain to provide another with callbacks, eliminating the need for polling, as we show in Section 4.2. By contrast, invoking a closure passed from an untrusted domain is unsafe in dynamic authorization browsers, because the closure might abuse the caller's privileges.

Dynamic authorization can be simulated in a static authorization browser by calling `eval` on string data received from another site, but the reverse is not true: static authorization cannot be easily simulated in a dynamic authorization browser.

## 3.3 Cross-domain frame access

Another security feature that differs across browsers is the access policy for the `frames` property of each window. In a web page that includes nested frames from various domains, sometimes two frames that are not in a direct parent-child would like to communicate with each other. Code running in one frame needs to navigate the frame hierarchy of the page in order to reach the other frame. We have observed several different browser policies enforced on this behavior:

- **Permissive.** Firefox and Safari will allow the frame structure of the page to be navigated so that the cross-domain frame can find another frame in the same domain.
- **Restrictive.** Opera does not allow access to the `frames` object of a cross-domain frame, preventing the frame structure of the page from being navigated.
- **Configurable.** Internet Explorer provides an advanced security setting called "Navigate sub-frames across different domains," which can be enabled, disabled, or set to prompt on every access. In Internet Explorer 6, this setting is enabled by default. When enabled, the browser behaves like Firefox and Safari; when disabled, it behaves like Opera.
- **Permissive, but restrict `location`.** Internet Explorer 7 also provides the "Navigate sub-frames across different domains," which is disabled by default. Unlike IE6, this setting does allow or prevent accessing the hierarchy of cross-domain frames in order to find a same-domain frame. Instead, it controls the browser behavior when Site A frame sets the `location` property of a Site B frame. When enabled, the location property may be set normally; when disabled, setting the location property causes a new window to open at that location. The change in Internet Explorer 7 has restricted some types of fragment identifier messaging. In any case, regardless of whether the setting is enabled or disabled, IE7 will allow the frame structure of the page to be navigated so that the cross-domain frame can find another frame in the same domain.

# 4. SINGLE WEB SERVICE

Our technique for passing data from the untrusted web service (e.g. www.webservice.com) to the mashup site (e.g. www.mashup.com) is to introduce a "throwaway" subdomain (e.g. webservice.mashup.com) that is used only to retrieve information from that web service. The user never sees this domain in the browser address bar, because it is used only by `IFRAME`s. These frames are structured such that data can be safely downloaded from www.webservice.com using a `<script>` tag, and none of the browser state associated with www.mashup.com (such as the user's authentication cookie, or the contents of a page) are ever accessible to www.webservice.com.

## 4.1 Setup phase

In order to create a Subspace channel between two sites, www.mashup.com and webservice.mashup.com, we perform a setup protocol that gives pages in both domains access to the same Subspace JavaScript object. This setup protocol (shown in Figure 3) is performed only once when the page is first loaded, and does not need to be restarted when further data requests are required.

1. **Create mediator frame.** Assume that the browser is at a location on www.mashup.com (the *top* frame). We create a hidden `IFRAME` (the *mediator* frame) pointing to a tiny page on www.mashup.com.
2. **Create untrusted frame.** Inside the mediator frame we create the *untrusted* frame pointing to a tiny page on webservice.mashup.com.
3. **Pass JavaScript communication object.** A Subspace JavaScript object is created in the top frame and passed to the mediator frame. Then the mediator frame and the untrusted frame change their domain to mashup.com by setting their document.domain variable. At this point, the mediator frame and the untrusted frame can communicate directly, because they both have the same document domain. The top frame and the mediator frame cannot directly communicate, because their document domains do not match. The mediator frame still has access to the Subspace object it obtained from the top frame, and passes this object

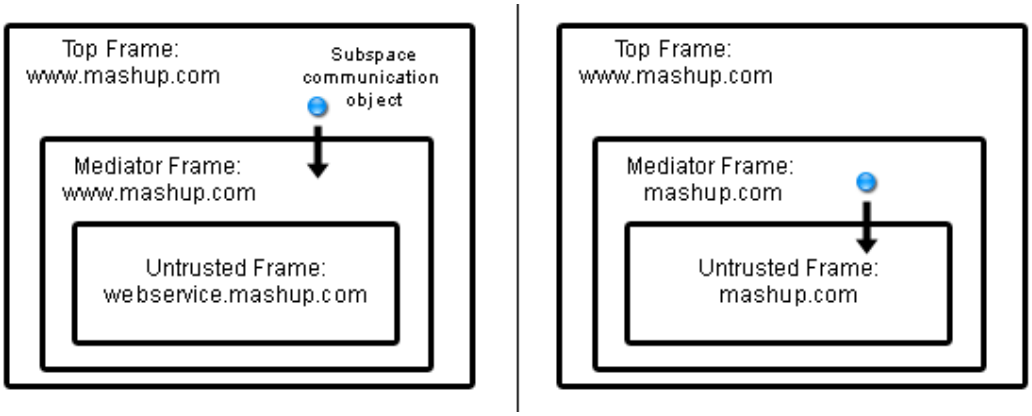


**Figure 3: Retrieving data from a single untrusted third-party domain. The first step (left) passes the communication object from the top frame to the mediator frame. The second step (right) can only occur after the mediator and untrusted frames change their document.domain to the suffix mashup.com.**

to the untrusted frame. Now the top frame and the untrusted frame have access to the same Subspace object, and can use it to pass arbitrary JavaScript data.

## 4.2 Data exchange

If the user's browser is Internet Explorer or Firefox, then it supports a static authorization model for closures described in Section 3.2. In this model, closures provide an easy communication mechanism between the top frame and the untrusted frame.

The untrusted frame adds a "data request" closure to the JavaScript object. This function takes a "data response" callback as an argument and is called when the mashup needs to request information from the web service. The top frame creates the data response callback, which takes the data as an argument and will perform whatever operations are necessary to respond to the completed request.

The data request function will dynamically insert a `<script>` tag into the untrusted frame, pointing to some data in JavaScript format that is hosted on www.webservice.com. When the `<script>` tag is done loading data from the remote web service, it invokes the data response callback to return that data to the top frame.

Unfortunately, Opera and Safari provide dynamic authorization for closures, so they do not support this callback system. The callback would run with the security privileges of the page that is calling it, not the security privileges of the page that created it. As a workaround for these browsers, we catch the security exception that is thrown when an unauthorized access occurs, and fall back on polling a property of the JavaScript object using the `setInterval` JavaScript function. Once data is ready, it can be read by the other party. This type of client-side polling does not involve any network requests, and it can be performed efficiently with a short waiting interval.

# 5. MULTIPLE WEB SERVICES

The scheme described in Section 4 is appropriate for the case where the mashup is only interacting with a single untrusted web service. If the mashup wants to interact with more than one web service or gadget, it not only needs to protect the security of its own domain, it also needs to keep these web services from compromising each other.

Unfortunately, because the untrusted frame for every web service lives in the mashup.com domain, an attacker's untrusted frame might be able to interfere with the untrusted frame of another web service, corrupting the Subspace channel and the data passed through it. Whether or not this issue is a problem is depends on the frame restrictions imposed by the browser, as described in Section 3.3. Table 1 shows the likely browser configurations and the server architectures that they can safely support.

## 5.1 Restrictive frame access

If the browser restricts access to cross-domain frames when navigating the frame hierarchy (Opera and some configurations of IE6), then the scheme described in Section 4 can be directly adapted to multiple untrusted web services. We simply create a new nested frame structure for each web service or gadget that needs to be included. The untrusted web services are nested inside sibling frames that cannot be accessed because the `frames` property of the main window is in the www.mashup.com domain and the gadgets are not.

## 5.2 Permissive frame access

In order to work properly for the majority of browsers, the mashup must also support the permissive frame access configuration of Firefox, Safari, IE7, and some configurations of IE6. This configuration makes separating gadgets much more difficult, because any frame anywhere on the page can be reached by any other frame, and if those frames are in the same domain, they can each access each other and intercept each other's communications.

In order to keep these frames from interfering with each other, we use a new throwaway domain for each web service that the mashup needs to interact with. For example, if the mashup needs to include two web services, it might use webservice1.mashup.com and webservice2.mashup.com.

Our solution, illustrated in Figure 4, is similar but subtly different from the frame structure described in Section 4.

1. **Create mediator frame.** Assume that the browser is at a location on www.mashup.com (the top frame).

| Browser Configuration | | | Supported Server Techniques | | | |
|---|---|---|---|---|---|---|
| Browser | Cross-domain frame access | Closure autho-rization | TUA + callback | TMU + callback | TUA + polling | TMU + polling |
| IE6 (default) | permissive | static | ✓ | | ✓ | |
| IE6 (restrictive) | restrictive | static | | ✓ | | ✓ |
| IE7 | permissive | static | ✓ | | ✓ | |
| Firefox | permissive | static | ✓ | | ✓ | |
| Opera | restrictive | dynamic | | | | ✓ |
| Safari | permissive | dynamic | | | ✓ | |

**Table 1: Browsers configurations and the server communication techniques they can safely support.**

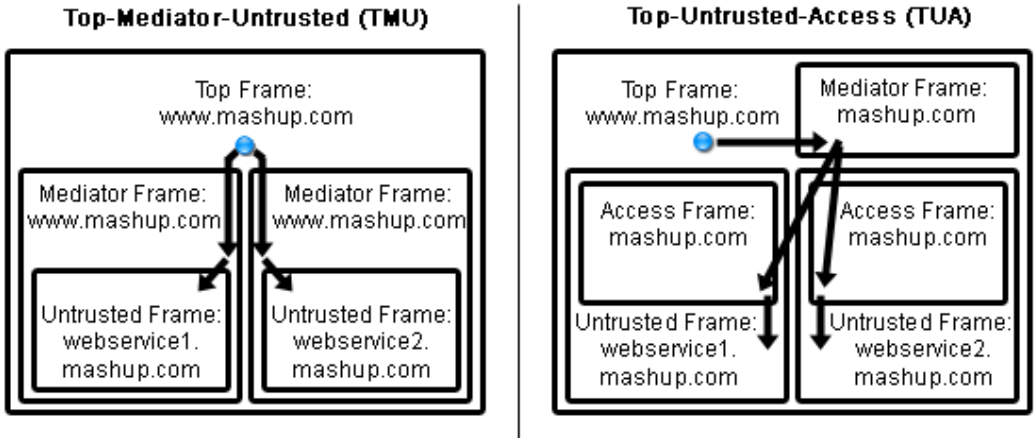


**Figure 4: Connecting to multiple web services. If the browser has a restrictive frame access policy, a top-mediator-untrusted frame structure should be used, but if the browser has a permissive frame access policy, a top-untrusted-access frame structure is required.**

We create an `IFRAME` (the mediator frame) pointing to a tiny page on www.mashup.com. As before, the mediator frame retrieves a JavaScript object from the top frame and changes its domain to mashup.com.

2. **Create untrusted frame.** As before, we create the untrusted frame pointing to webservice1.mashup.com, but rather than putting it inside inside the mediator frame, we put it inside the top frame. Thus, the mediator frame and the untrusted frame are siblings.

3. **Create access frame.** Next, we add the *access* frame inside the untrusted frame. The access frame obtains a "container" JavaScript object from the untrusted frame, and then it changes its domain to mashup.com.

4. **Pass JavaScript communication object.** Due to the browser's permissive frame access policy, the access frame can get a handle on the mediator frame, and because they are both in the mashup.com domain, the access frame can obtain the Subspace object from the mediator frame. The access frame puts this object into the container it shares with the untrusted frame.

5. **Cleanup.** At this point, the untrusted frame disposes of the access frame, which is no longer needed. The untrusted frame has the Subspace object it needs to communicate with the top frame.

6. **Repeat for every gadget.** The process is repeated for every other untrusted web service or gadget that needs to be included. At the end, the mediator frame is no longer needed and can be disposed.

7. **Load untrusted content.** At this point, all the gadgets have a Subspace communication channel to the top frame, but none of them have access to each other. Since the setup phase is complete, the top frame can safely issues the command to load the untrusted content, such as cross-domain `<script>` tags, into each of the frames.

## 5.3 Setup Integrity

The setup mechanism described in Section 5.2 relies on the assumption that the untrusted code does not run until all Subspace channels have been securely initialized. Setup integrity is not necessary for protecting the user's authorization credentials at the gadget aggregator site, but it is necessary for protecting each individual gadget's communications with the parent page from interference or interception by other gadgets.

An example attack on setup integrity would be for webservice1.mashup.com to open a popup window and stash some malicious code there to retain control over the browser session. The attacker would then restart the initialization process in the original window by resetting its location, making it appear to the mashup site that the user is arriving at the site for the first time:

```
window.opener.location = "about:blank";
window.opener.location = "http://www.mashup.com/";
```

During the second initialization process, the popup could try to find the mediator and access frames and maliciously modify them, potentially corrupting the Subspace channel.

To prevent this class of attacks, a mashup could ensure that the domain used to communicate is different from one page load to the next. On the second visit, the main frame would be located at a different domain, e.g. www.2.mashup.com and the untrusted web services would be located at webservice1.2.mashup.com and webservice2.2.mashup.com. The number of visits would be tracked in a session cookie that expires when the browser is closed. Also, the gadget aggregator would need to reload the page to restart the setup process if more web services or gadgets need to be added beyond the ones that were created during the first setup phase.

Another (more cumbersome) approach to solve this problem would be to use public key cryptography to protect communications between gadgets over the Subspace channel.

To summarize, setup integrity enforcement is only necessary for mashups with untrusted gadgets from more than one source, where the browser has a permissive frame architecture, and gadgets are communicating client-side information with their parent that needs to be protected from interception or interference by other gadgets. For gadgets that only communicate non-sensitive information (e.g. desired height) on the client side, setup integrity enforcement is not required.

# 6. EVALUATION

In order to evaluate the performance impact of Subspace, we performed some timing measurements using a simple mashup and a prototype gadget aggregator.

## 6.1 Mashup Measurements

Our example mashup, called KittenMark, shows a list of the 20 most recent kitten photos from the Flickr photo sharing site [4] and allows the user to post them to the del.icio.us bookmarks site.

We tested three site architectures for this mashup:

- **Proxy.** We built a proxy that connects to the Flickr web service to get the list of kitten photos, and relays this data back to the mashup. The mashup connects to this proxy safely using an XMLHttpRequest and uses the response data to populate the page.
- **Unsafe.** We built another version of the mashup that downloads the data directly from the Flickr web service using a cross-domain <script> tag. This approach is unsafe if the Flickr web service is not trusted, because it allows Flickr to observe or hijack the user's KittenMark session.[2]

[2]Our choice of Flickr as an example untrusted web service is not meant to disparage Flickr in any way.

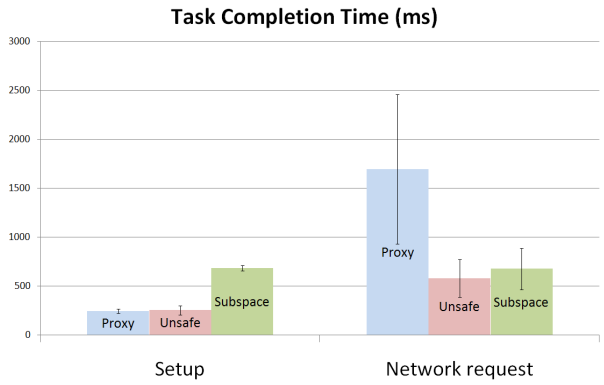


**Figure 5: For the mashup experiment, Subspace took longer to set up, but its network requests were faster than the proxy approach.**

- **Subspace.** Finally, we built a mashup using the web service architecture for Subspace described in Section 4. This approach also makes network requests using a cross-domain <script> tag, but the browser same-origin policy prevents Flickr from accessing users' KittenMark accounts.

In order to test the speed of these three approaches, we built an automated timing framework in JavaScript that measured the time to load the initial page and the time to download the latest list of kittens from Flickr. The Flickr web service is hosted in Sunnyvale, California, and our mashup server was hosted in New Haven, Connecticut. To simulate an initial page load, we bypassed caching by appending a random unique identifier to the query parameter of all URLs. Our client machine was located in Stanford, California, with the IE7 browser and a broadband internet connection. We performed 25 trials with each architecture.

Our results are summarized in Figure 5. We found that Subspace took slightly longer during the initial setup process, because of the time required to load the hidden IFRAMEs. Once the setup process was complete, network requests were faster than the proxy approach and comparable to the cross-domain <script> tag approach.

## 6.2 Gadget Aggregator Measurements

Our second experiment involved building a simple gadget aggregator that allows the user to customize the font color of all his or her gadgets. We tested three gadget aggregator architectures:

- **Sandboxed.** We used a third-party iframe approach that reloaded the gadget whenever the user's desired font color changed.
- **Unsafe.** We built another version of the aggregator that included the gadget's source code inline with the page. It used JavaScript to pass the desired font color to the gadget region of the page. This approach is unsafe if the gadget author is not trusted, because it allows the gadget full access to the surrounding page.

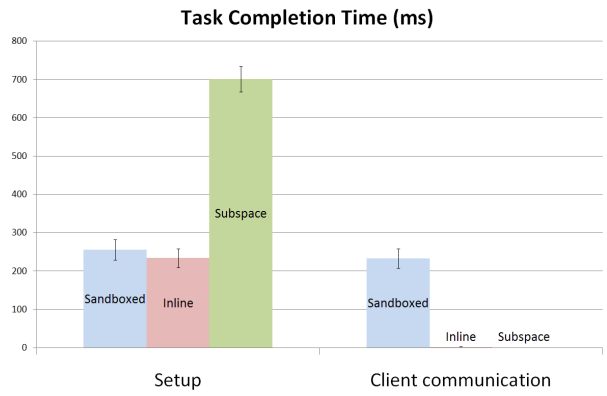


**Figure 6: For the gadget aggregator experiment, the initial page load took longer with Subspace than with the sandboxed and inline configurations. However, Subspace allowed instantaneous communication without the security disadvantages of inlining.**

- **Subspace.** Finally, we used Subspace to pass the desired color information from the parent page to the untrusted gadget frame, in JavaScript.

To measure the setup time and the time it took to change the font color, we used the same browser and automated timing methodology from Section 6.1. Our results are summarized in Figure 6. We found that for the initial page load, the sandboxed and unsafe approaches were faster than Subspace, because fewer frames were required. However, when responding to a font color change request, the inline and Subspace architectures made the change almost instantaneously, while the sandboxed approach required the user to wait for the page to load.

Because IE7 has a static authorization mechanism as described in Section 3.2, our gadget aggregator was able to use callbacks for optimal client communication speed. With a dynamic authorization browser such as Opera, polling would add a small amount of delay to client communication, perhaps 20 milliseconds or more depending on the polling interval.

# 7. DISCUSSION

## 7.1 Subdomains

Subdomains are easy to acquire, and it is straightforward to configure a DNS server to have an arbitrary number of subdomains pointing at the same server. For websites with restrictive hosting conditions, for example using Geocities, setting up subdomains may prove to be difficult. For these types of low-budget sites, other types of cross-domain data exchange might be more appropriate.

Some sites may wish to remove the "www" from the beginning of their name, for example mashup.com instead of www.mashup.com. Although it might seem that decision is incompatible with the subdomain communication scheme presented here, in fact it poses no serious problems. Browsers treat the fully qualified mashup.com domain differently from the domain suffix mashup.com. Somewhat unintuitively, the fully qualified domain page can switch into the untrusted domain suffix mode by running this command:

```
document.domain = document.domain;
```

Although this command would seem to be a no-op, in fact has a dramatic effect on the security of a frame, allowing it to access and be accessed by any subdomain of mashup.com. The top frame would need to avoid using this command.

## 7.2 Limitations

One limitation of Subspace (and existing gadget aggregator sites, which rely on cross-domain frames) is that the frames can launch a denial-of-service attack on the browser. For example, a misbehaving web service might navigate the browser away from the mashup site, or display an endless chain of alert dialogs, preventing the user from using the site. Because these behaviors are relatively easy to detect and do not pose a privacy threat, we consider them to be merely an annoyance. These behaviors may be preventable using some of the emerging technologies discussed in Section 8.

Another possible concern is that the untrusted data source or gadget would pop up a new window asking the user for their authentication credentials. For this reason, it is important that the subdomain be named in such a way that a user would be able to clearly identify the web service that controls it. Most modern browsers display the source of the popup at the top of the window regardless of whether the site creating the popup requests that the information to be displayed.

## 7.3 Input Validation

Many of the techniques described here have applications for so-called "cross-site scripting attacks" that exploit a lack of proper input validation in websites. These attacks should not be confused with legitimate uses for transferring of data between scripts of different sites. In a cross-site scripting attacks, the attacker often needs to pass data across domain boundaries in order to drive the user's browser and retrieve stolen information, but the root cause of the vulnerability lies in poor input validation. Securing the site against these attacks is a prerequisite for applying the security techniques discussed in this paper.

# 8. RELATED WORK

Subspace is built on the existing features of the current generation of web browsers. Several proposed web standards hold the potential to deliver built-in cross-domain data exchange features to future web browsers. These standards could make mashups even easier to build, but cannot be relied upon until implemented by all major browser vendors and installed by most users.

## 8.1 XML access-control instruction

A W3C working draft proposes a new processing instruction, <?access-control?>, for authorizing read access to XML content [12]. For example, the National Oceanic and Atmospheric Administration may declare that their XML weather data can be accessed by any application, while a stock ticker provider could allow access only to an individual partner site mashup.com that has licensed that data:

```
<?access-control allow="*.mashup.com"?>
```

This processing instruction would be placed in the document's XML prolog and specifies who can access the document. In this example, the directive would be similar (although not exactly equivalent) to stockticker.com setting its `document.domain` to mashup.com, a command that is denied by all current browsers for security reasons. Although these restrictions prevent web developers from accidentally building insecure websites, they also prevent cross-domain access control policies from being expressed. The <?access-control?> instruction could provide this expressive power, but unfortunately it is not available in current browsers.

## 8.2 JSONRequest

An alternative to XML is JavaScript Object Notation (JSON), a compact data representation that is designed to parse as valid JavaScript. Data in JSON format can be interpreted as script the browser's JavaScript interpreter, but this approach is not safe because the data may contain malicious active code. The proposed JSONRequest browser object [3] addresses this concern by parsing the JSON data without interpreting it as code.

The JSONRequest browser object would be capable of performing cross-domain data requests, unlike the XMLHttpRequest object. It would accept only data with a mime type `application/json`, ensuring that it cannot be used to access web pages or web services that were not designed specifically with JSON in mind. Unlike XMLHttpRequest, the JSONRequest does not send cookies.

## 8.3 BrowserShield

Although JSONRequest and the access-control directive can allow safe cross-domain network requests, they do not provide the controlled cross-domain client-side communication necessary for gadget communication. One alternative to cross-domain gadget `IFRAME`s is to preprocess the gadget's JavaScript code to ensure that it can only perform actions within a set of acceptable guidelines, then run it in the same domain as the mashup site. The BrowserShield framework has the potential to perform this code transformation, either at runtime on the client side or as a one-time transformation on the mashup server [10].

Developing a complete set of BrowserShield sandbox policies is a challenging problem, but it would only need to be solved once by the maintainers of the BrowserShield library, rather than individually by each website. Furthermore, BrowserShield could prevent some denial-of-service behaviors that are allowed by cross-domain frames, such as navigating the parent frame to a new location or popping up an endless sequence of alert dialogs.

## 8.4 Cross-document messages

Cross-document messages [6] are a proposed browser standard that would allow frames to send string messages to each other regardless of their source domain. To send a message, a frame could call the `postMessage(data)` method of the cross-domain frame. To receive messages, a frame would observe the "message" event:

```
document.addEventListener('message', handler, false)
```

The `handler` function would be able to check the domain of the message before reading to message's data to ensure that the message is coming from an authorized source.

Although the current cross-document message proposal does not support the passing of data types other than strings, this limitation could be circumvented through the use of serialization and deserialization library functions.

This primitive would be well-suited to providing the client-side communication capabilities of Subspace without requiring the use of additional subdomains. We hope that the cross-document messages proposal, or a similar one with richer data type support, will eventually be adopted by all browsers.

# 9. CONCLUSION

Web mashups have created a new generation of wildly popular and successful web services, marking a paradigm shift in web service development. However, web mashups also require a departure from same-origin policy enforced by browsers. Current practices in achieving cross-domain communications are either insecure, inefficient, or unreliable. In this paper, we have presented Subspace, a cross-domain communication primitive that allows efficient communication across domains while still sandboxing untrusted code. Our primitive uses existing browser features as building blocks and is therefore is highly practical. Our prototype implementation of Subspace is compatible with all major browsers. We look forward to using the new mashups that Subspace will enable.

# 10. REFERENCES


[1] Vikram Agrawal. TODO List. `http://googlemodules.com/module/612/`.

[2] Richard Cornford. JavaScript Closures, March 2004. `http://jibbering.com/faq/faq_notes/closures.html`.

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

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

[5] C. Fournet and A. D. Gordon. Stack Inspection: Theory and Variants. In *Symposium on Principles of Programming Languages*, 2001.

[6] Web Hypertext Application Technology Working Group. Web Applications 1.0, February 2007. `http://www.whatwg.org/specs/web-apps/current-work/`.

[7] ECMA International. Standard ECMA-262, December 1999.

[8] C. Jackson, A. Bortz, D. Boneh, and J. Mitchell. Protecting Browser State Against Web Privacy Attacks . In *Proc. WWW*, 2006.

[9] T. Powell and F. Schneider. *JavaScript: The Complete Reference*. McGraw-Hill/Osborne, second edition.

[10] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML . In *Proc. OSDI*, 2006.

[11] J. Ruderman. JavaScript Security: Same Origin. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[12] W3C. Authorizing Read Access to XML Content Using the <?access-control?> Processing Instruction 1.0. `http://www.w3.org/TR/access-control/`, May 2006.