Journal of Web Engineering, Vol. 12, No.3&4 (2013) 291-316

A SECURE PROXY-BASED CROSS-DOMAIN COMMUNICATION FOR WEB MASHUPS

SHUN-WEN HSIAO[1,2] YEALI S. SUN[1] MENG CHANG CHEN[2]

National Taiwan University[1]

{r93011,sunny}@im.ntu.edu.tw

Academia Sinica[2]

{hsiaom, mcc}@iis.sinica.edu.tw

Received October 28, 2011
Revised March 13, 2013

A web mashup is a web application that integrates content from heterogeneous sources to provide users with an integrated and seamless browsing experience. Client-side mashups differ from server-side mashups in that the content is integrated in the browser using the client-side scripts. However, the legacy same origin policy implemented by the current browsers cannot provide a flexible client-side communication mechanism to exchange information between resources from different sources. To address this problem, we propose a secure client-side cross-domain communication mechanism facilitated by a trusted proxy and the HTML 5 postMessage method. The proxy-based model supports fine-grained access control for elements that belong to different sources in web mashups; and the design guarantees the confidentiality, integrity, and authenticity during cross-domain communications. The proxy-based design also allows users to browse mashups without installing browser plug-ins. For mashups developers, the provided API minimizes the amount of code modification. The results of experiments demonstrate that the overhead incurred by our proxy model is low and reasonable. We anticipate the proxy-based design can help the mashup platform providers to provide a better solution to the mashup developers and users.

Key words: Web mashup, same origin policy, access control, proxy
Communicated by: M. Gaedke & T. Tokuda

1 Introduction

A *web mashup* is a web application that integrates content or services from multiple sources. By combining content from different websites, a web mashup can provide users with an integrated and seamless browsing experience on a single web page. The web developers can easily incorporate existing web services provided by third parties into one web mashup to enrich the web applications. The concept of the web mashup makes the web application developers create more creative web services for the users.

For a *simple mashup*, it looks like a dashboard that collects information from different sources. It simply displays the content retrieved from different providers on a single web page. For example, it can display contents of a remote RSS/Atom feed, or contents of the web's most visited social news sites and portals. One example of such mashup is PopUrls (http://popurls.com/), which integrates the

contents of the popular sites and portals, including Twitter (http://www.teitter.com/), del.icio.us (http://del.icio.us/), Digg (http://www.dogg.com/), and Flickr (http://www.flickr.com/), and then it displays all the headlines from these sources on a single web page. With such design, a user can retrieve all the information at once without opening multiple web pages.

In *interactive mashups*, the content of one source may be able to communicate with the content of the other sources to provide extra functionalities and better interactions with the user. HousingMaps [1] is an example of this type of mashup. It combines the property database of the Craigslist website (http://www.craigslist.com/) with the map data from Google Maps (http://maps.google.com/) on an integrated page. A user can employ HousingMaps to search for houses listed in Craigslist. Then, the respective locations and information about the properties will be plotted on the Google Maps. In this case, the information (i.e., the search options of the property, the search result and location details) needs to be transmitted between different sources via the browser by mashup's client-side scripts. Another popular example of interactive mashups is content-aware ads. Ad scripts that dynamically fetch ads' content from ad networks (e.g., Google AdSense) may need to retrieve the content of the web page before the ads are displayed on the web page.

As threats to privacy may occur when information is exchanged between different sources, browsers currently implement the *Same Origin Policy* (SOP) [2], which controls who has access to information carried by the browser. The SOP is an *all-or-nothing* trust model. Documents from the same source are allowed to access each other's contents, but documents from different sources do not have any access rights. However, such inflexible policy limits the development of interactive mashups, so the legacy SOP forces the web developers to make trade-offs between security and functionality.

Various techniques have been developed to perform cross-domain communication in mashups to avoid being blocked by the SOP. For example, some mashups use a shared HTML property, i.e., `window.location`, as a channel to exchange messages; while others (e.g., [3]) introduce new HTML tags and browser plug-ins. Another method, [4], deliberately modifies the source value (i.e., `domain`) of the documents so that these documents can communicate with each other under the SOP.

For mashups to work effectively, we argue there should be a flexible and fine-grained access control model. The model should satisfy the following principles of security and in the meantime provide enough flexibility for the mashup developers. (a) *Confidentiality*: In mashups, a user's data should only be available to an authorized source. (b) *Integrity*: A user's input on a web page should not be corruptible by distrusted parties. (c) *Authenticity*: The entity that receives the cross-domain data should be able to validate the identity of the sender. (d) *Flexibility*: Mashup developers should be able to define trust relationships in a more fine-grained manner, not in an all-or-nothing fashion.

The contributions of our works are as follows. (1) We build up a client-side cross-domain communication library on the top of the HTML 5 `postMessage` method with a proxy-style fashion. It provides the developers a convenient, flexible and secure way to implement interactive mashups. (2) Our solution can automatically generate/enforce fine-grained and secure access logic (using JavaScript) based on XML-based access control policy (ACP) provided by the mashup developers. No need to add new HTML tags or install browser plug-ins. (3) We implement a prototype system and examine the overhead of the proxy. The result shows the overhead is linear to the number of shared components. (4)

Our design provides the mashup platform providers with an easy way to cooperate with other web services, while simultaneously protecting the private data of end users via the access control policy.

The remainder of the paper is organized as follows. In Section 2, we provide some background information about mashups. Section 3 contains the description of the proposed model. We present the prototype system and its implementation in Section 4, and evaluate our design in Section 5. In Section 6, we discuss existing practices and communication mechanisms for web mashups. Section 7 contains some concluding remarks.

2 Background

2.1 Web Mashups

There are several roles in web mashups. An *integrator* is a site that hosts web mashups, and a *provider* is a site that provides content or service to web mashups. A *mashlet* is client-side content from the provider, and it usually includes a piece of active content and client-side script. When a user requests a web page from the integrator, the retrieved web content, excluding the embedded mashlets, is called the *original content*.

Figure 1 shows an example of mashups with one original content (the integrator site is http://www.housingmaps.com/), one map mashlet (the map provider site is http://maps.google.com/), and one house mashlet (the house provider site is http://www.craigslist.com/). This mashup has three blocks: the biggest block with several control options is from the integrator; the left-hand-side mashlet is from the map provider; the right-hand-side mashlet is from the house-listing provider. The integrator combines two other services to provide the user with a more convenient way to search house information. Usually, a mashlet can be embedded into a mashup by HTML tag `<IFRAME>` or `<DIV>`. A mashlet may contain JavaScript code that enables it to communicate with other mashlets, manipulate the content or interact with the user. Note that the access policies between the original content, the mashlet itself and other mashlets are enforced by the SOP.

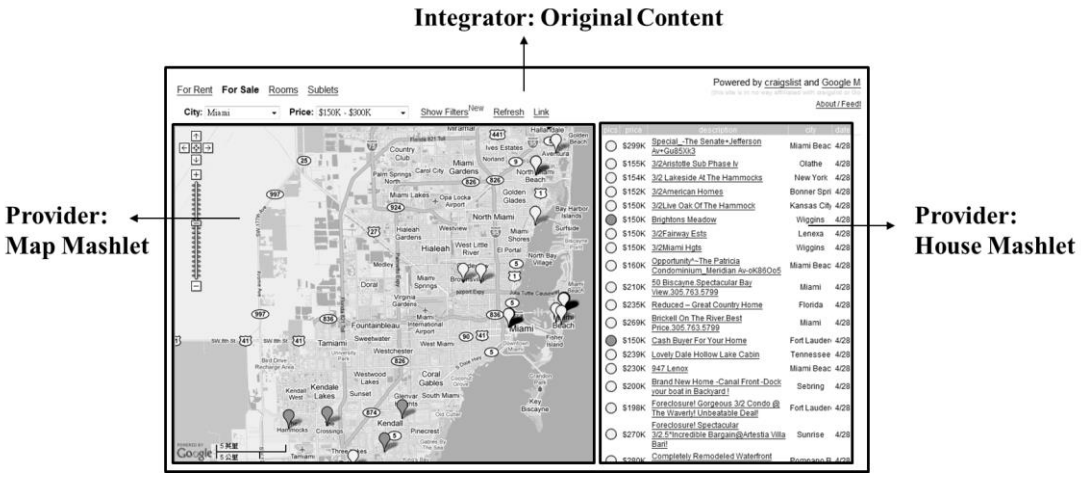

Figure 1 A web mashup example: HousingMaps.com.

At the browser side, an HTML document and all its elements is transformed into a *Document Object Model* (DOM) at runtime. Every HTML document (including the original content and the mashlets in `<IFRAME>`) are loaded into a DOM container, called `window`. The document itself is stored in an object named `window.document`. DOM is the standard platform-independent and language-neutral programming interface used by client-side scripts to access and manipulate the content, structure and style of the documents. Under the SOP, documents from different sources cannot access with each other. The issue of the mashups we address is how to design a flexible and fine-grained access control mechanism under the SOP between HTML documents from different sources.

2.2 Mashup Architectures

Server-side mashups (figure 2a) integrate data from different sources at the server and return the aggregated page to the client. The integrator acts as a data aggregator. It appears to the browser that all the content is from the same origin, i.e., the integrator. Yahoo Pipes (http://pipes.yahoo.com/pipes/) utilizes this design. When a user visits the mashup created via Yahoo Pipes, the browser connects to pipes.yahoo.com to get data. The request is redirected by Yahoo to one or more providers, like The New York Times, to retrieve data. Yahoo Pipes then aggregates the response data collected from different sources into one single web page and then passes it from pipes.yahoo.com to the browser for display. From the perspective of a browser, there is only one source (or domain) of these contents. Hence, there has no the cross-domain communication problem at the client side.

However, the integrator is responsible for collecting data; the user needs to delegate authority to the integrator to obtain content from the providers. It requires the clients to trust the integrator. Such approach may possess privacy and security concerns. Other drawbacks of server-side mashups include additional processing delays and limited scalability. The latency is caused by making one additional round trip between the client and integrator every time the user makes a request to the mashup.

Client-side mashups (figure 2b) differ from server-side mashups in that the integration of content or services takes place on the client side, i.e., the browser. The user requests the service or content from the provider directly. With client-side mashups, the user does not need to place so much trust in the integrator. However, in this case, it is necessary to consider the problem of cross-domain communication, since the original content and the mashlets usually come from different sources. A client-side mashup reduces the amount of trust that the user has to place on a distrusted third-party integrator. Hence, it does not possess the aforementioned drawbacks of a server-side mashup.

Our design combines the advantages of server-side and client-side mashups. We propose a secure client-side communication scheme that is facilitated by a trusted proxy (figure 2c). The content from the integrator and providers will be integrated and displayed at the browser as usual (which is similar to a client-side mashup), but these returned web pages are annotated by our proxy (which is similar to a server-side mashup). The proxy will automatically capture all the outgoing web requests and incoming responses. It then downloads the access control policies (ACP) specified by the integrator and providers to generate a site-specific access control policy for this mashup. According to the site-specific ACP, the proxy generates and inserts corresponding client-side JavaScript snippets into the returned web pages. Every time when a cross-domain communication request is issued at the browser, the inserted JavaScript snippets will provide a secure communication channel for the mashlets and enforce the providers' security settings. Therefore, the security is guaranteed.

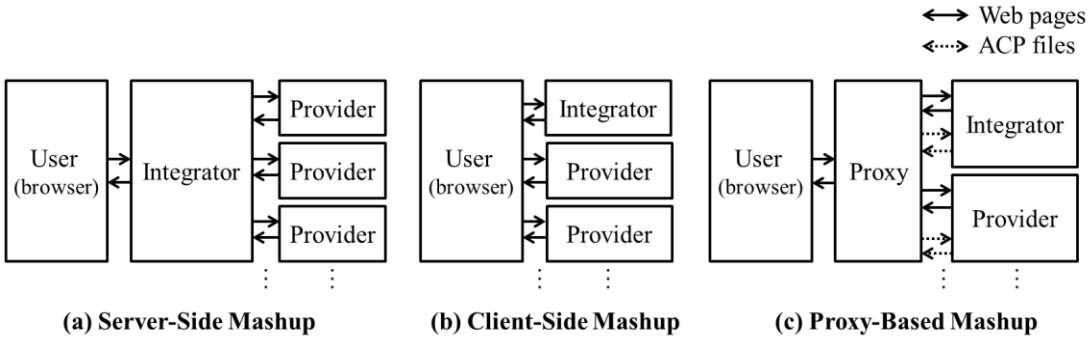

Figure 2 Two types of mashups and our proposed proxy-based scheme.

2.3 Same Origin Policy

The Same Origin Policy (SOP) is enforced by the browser. It is assumed that two web pages derive from the same *source* (or origin) if the application layer protocol, port and domain name are the same for both pages. Recall that the SOP is an *all-or-nothing* mechanism. For a server-side mashup, all the contents are from the "same source", i.e., the integrator, so there is an *all-trust* relationship between the mashlets (no matter these mashlets are embedded by `<DIV>` or `<IFRAME>`). Hence, a mashlet can retrieve the user's information without authority. In contrast, in a client-side mashup implemented by `<IFRAME>`, mashlets derive from different sources in a *no-trust* relationship. The `<IFRAME>` tag has an attribute named `src`, and it specifies the URL of the mashlet document to embed in the mashup. As mentioned, if the contents come from different source, they run in an isolated environment (in practice, they belong to different `document` objects in DOM) with no access to each other.

In terms of access control within the browser, the *all-or-nothing* trust model does not provide any flexibility for web mashups. The legacy SOP forces web mashup developers to make trade-offs between security and functionality.

2.4 HTML 5 postMessage Method

HTML 5 specifies an API, named `postMessage`, for asynchronous communication between DOM `window` object. The `postMessage` API was originally implemented in Opera 8 and is now supported by modern browsers, such as Internet Explorer 8, Firefox 3, and Safari. To send a message, a `window` object needs to invoke the `postMessage` method of its target window; and to receive messages, a `window` needs to register an event handler to catch the triggered event, called a `message`. The invocation of the `postMessage` is not restricted by the SOP so that cross-domain communication is possible with this new method. For security concern, the self-defined event handler may check the source of the message to ensure it comes from an authorized source. However, such authentication mechanism is not a default functionality provided by the `postMessage`. However, Barth *et al.* [7] demonstrate an attack on the channel's confidentiality using frame navigation. In light of this attack, the `postMessage` channel lacks confidentiality.

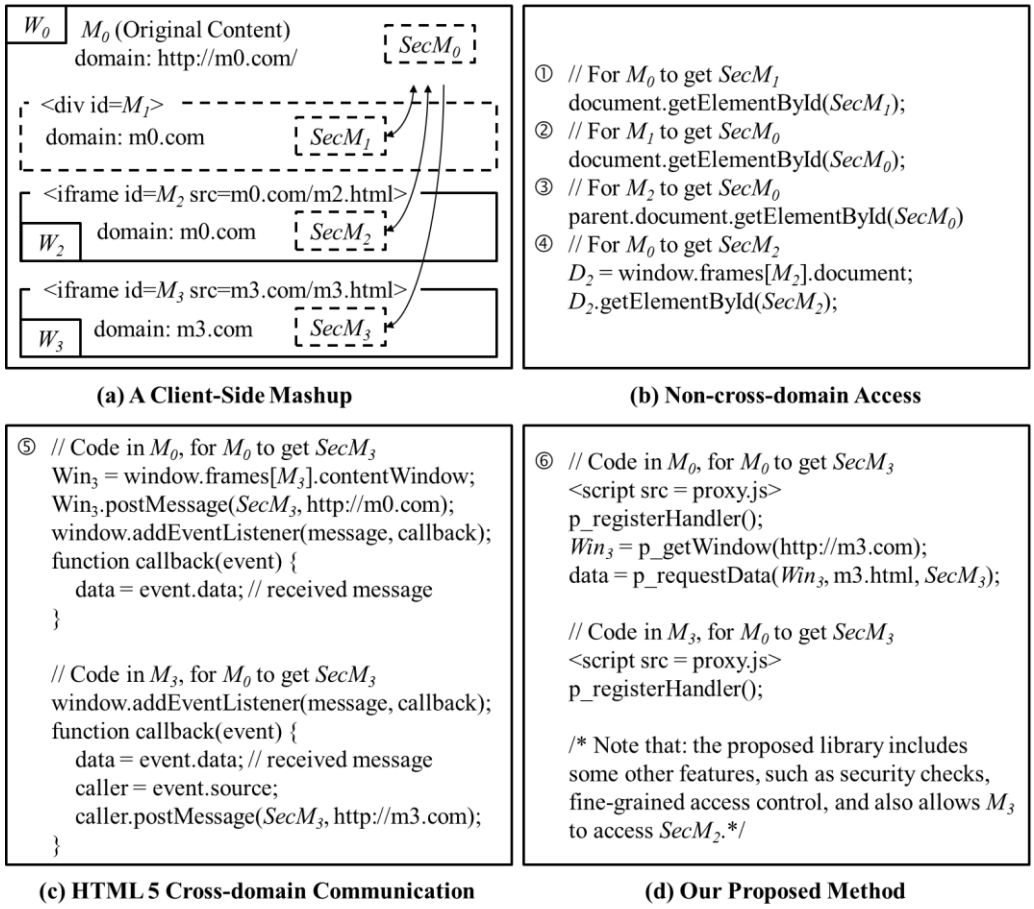

Figure 3 The example of conventional object access and our proposed method in client-side mashups.

In figure 3, we show an example of `postMessage` usage with a client-side mashup under the SOP, as well as our proposed method. Figure 3a is a simple mashup with the original content and three mashlets. In this example, the original content (M_0), mashlet M_1 and mashlet M_2 are from the same domain (e.g., http://m0.com/) and the content of the last mashlet M_3 are provided by another domain (e.g., http://m3.com/). M_1 is embedded in the original content by `<DIV>`, while M_2 and M_3 are embedded in the M_0 by using `<iframe>`. (Note that there is no way to use a `<DIV>` to embed a mashlet from other domain in a client-side mashup.) Assume there is a secret object named $SecM_x$ in each entity M_x, respectively. For example, it could be a secret web cookie provided by the service domain. In figure 3b, we show the JavaScript code of non-cross-domain access. Under the SOP, M_0 and M_1 can access each other's object without any restriction (i.e., *all-trust*) by simply using JavaScript function `document.getElementById()`,since M_0 and M_1 are in the same document. On the other hand, M_0 and M_2 are not in the same document, because the browser would create another `document` for M_2 due to the `<IFRAME>` tag. However, the `document` of M_2 is a child node of `document` of M_0. Therefore, in order to access the secret object $SecM_0$, M_2 can call the JavaScript function `parent.document.getElementById('SecM0')` to obtain the object. On the other hand,

since M_0 and M_2 are from the same origin, M_0 is able to obtain M_2's document object by using `D2 = window.frames['M2'].document` and further gets $SecM_2$ by using JavaScript function `D2.getElementById('SecM2')`.

However, for M_0 and M_3, it is not that easy. Under the SOP, M_0 and M_3 are *no-trust* so that conventional JavaScript calls cannot cross the access boundary. To perform such cross-domain communication, we use figure 3c to show a detail example of HTML 5 `postMessage` method for cross-domain access. For M_0 to get $SecM_3$, M_0 should first obtain M_3's window, (for instance, `win3 = window.frames['m3'].contentWindow`), and then M_0 invokes the `poseMessage` method `win3.postMessage('SecM3', 'http://m0.com/')` to send a message. Note that the first parameter is the message (string) sent to M_3 and the second parameter is the domain of M_0. To receive such request, M_3 should register an event handler to its `window`. Due to different browser implementations, it could be `window.addEventListener('message', callback)` or `window.attachEvent('onmessage', callback)`. The first parameter is the event type to listen to and the second parameter is the function name of the event handler. The handler function will be called once the `postMessage` event (i.e., `message` or `onmessage`) is received by the callee (i.e., M_3) from the caller (i.e., M_0 in this case). In the example, an object named `event` stores all the information sent from the caller. In practice, the `event.data` usually carries the request of retrieving object $SecM_3$. Then, M_3 can reply such request by invoking another `postMessage` by using `event.source.postMessage('SecM3', 'http://m3.com/')`. The `event.source` is the `window` object of the caller (i.e., M_0's `window`), so that it is convenient for M_3 to send a message back to its caller. In the example, a string `SecM3` is sent back. As for M_0, it should register a handler function for receiving the returned message as well. In this way, M_0 and M_3 can exchange messages between each other.

As for M_3 to retrieve an object of M_0, M_3 should first get the `window` of M_0 and then perform the `postMessage` procedure. However, we point out that getting any object of M_0 (e.g., $SecM_0$, $SecM_1$, and $SecM_2$) is not easy for an embedded M_3 with `<iframe>`. Not only because they are in *no-trust* relation under the SOP, but also the embedded mashlets may not know the structure the mashup. It is difficult for the mashlet developer to communicate with other mashlet or the original content without any help. Hence, we solve these problems in our proposed library.

It seems convenient to perform cross-domain communication via the `postMessage`. However, some issues should be addressed in practice. First, the `postMessage` receiver needs to validate the message on its own to ensure it is from an authorized source (domain). The mashlet programmer needs to write a script (the length of the script depends on how many domains it trusts) for validation. Second, the trust relationship between mashlets may change dynamically. Such validation scripts may not be scalable, and it should be modified timely. Third, basically a mashlet has no information about the mashup and its sibling mashlets. A mashlet may not know what else mashlets are embedded in the same web mashup. However, for the purpose of inter-mashlet communication, the message sender needs to get the receiver's window id to invoke the `postMessage` method. In practice, it is difficult and not secure for the sender to search for receiver's information in integrator's DOM. Moreover, the SOP forbids such information access if the sender and the original content do not have the same origin. In this case, without the help of integrator, it is nearly impossible for a "blind" mashlet to find the frame of its target mashlet.

The HTML 5 `postMessage` method is a nice solution to cross-domain communication problem. However, we argue that the mechanism is too simple. It lacks of security checks and fine-grained access control. The mashup developers have to implement these functionalities on their own. In this case, we believe providing a well-developed library to perform these underlying features would be a benefit for the mashup platform providers.

In our proposed system, we take the advantage of the `postMessage`, and we provide a proxy-based solution to mitigate the issues mentioned above to allow fine-grained, efficient and secure interactions between mashup components. Figure 3d shows the example code of invoking our proposed proxy-based cross-domain library. First, the proxy will automatically insert the `<script src="proxy.js">` in the mashlets to enable our libraries. The `p_registerHandler()` will be called automatically, once a mashlet is completely loaded in the browser (i.e., `window.onload`) to deal with the aforementioned registration procedure. Until now, the mashlet developers do not have to modify any of their codes. If M_0 (or M_2) needs to access certain objects of M_3, the developers can simply call our library `p_queryWindow(aURL)` to get the window of M_3 by given M_3's domain or URL for reference. And then by calling our wrapper function `p_requestData(targetWin, targetURL, elementID)`, the value of object `elementID` in M_3 will be returned. In this case, the cross-domain communication is completed. The mashlet developer can also specify which domain can or cannot access its private object by a XML policy file. Our library will check it before returning. We will discuss the detail implementation of `p_queryWindow` and `p_requestData` later.

3 System Design

3.1 Security and Functional Requirements

Confidentiality. If a client-side communication mechanism is implemented, the message can only be seen by the nominated parties. Sometimes, a secret symbol is shared by the user and the original content/mashlet for authentication purposes. One scenario that violates confidentiality is when a user views a page with a secret web cookie or password. Then, a malicious mashlet might read it and send it to a remote host. A secure design must ensure that the mashlet can only see the proper content.

Take another example, Gmail provides advertisers with a service which Google dynamically selects advertisements (displayed on the right-hand side of the web page) most relevant to the content of the email content. If Gmail is designed as a client-side mashup architecture (actually it is not), then we can view each ad as a mashlet and the email content as the original content. A secure and flexible browser-side communication needs to make sure that the ads mashlet can only see the content of the email but cannot see any other data in the original content, such as the contact list.

Integrity. Malicious parties must be prevented from accessing exchanged messages. For example, Twitpay.com is a web mashup application that allows people to send small payments through Twitter and PayPal. To do this, they include the recipient's username and the amount in the exchanged message. There could be a security problem if third-party ads are mashed up on the same web page. The ads mashlet must not be able to tamper with the value of the payment in the user's input data.

Authenticity. The communication mechanism should guarantee the identity of the data sender as well as the data receiver. Malicious mashlet can act as others to obtain the data that it cannot retrieve.

In addition, it should ensure that the communication parties comply with the ACP set by the mashup developers. That is, only nominated parties can access the shared data.

Flexibility. We believe the ACP should be more fine-grained to the *element level*, rather than domain level. A mashlet may want to share an element to some authorized mashlets, but prevent access by others. Therefore, in our design, the ACP is linked to an element. In our case, under the SOP, a mashlet may not access another mashlet's element due to different origins, but it can specifically ask for the value of the element using our proposed mechanism if it passes the trust validation.

3.2 The Proposed Web Mashup Model

We argue the legacy SOP restriction is insufficient to support the various trust relationships for data access that are desired by mashup developers. We anticipate that a mechanism that provides "content isolation" and "secure communication" are necessary. The mechanism needs to allow the original content and mashlets to communicate with one another flexibly in client-side mashups and guarantees confidentiality, integrity, and authenticity at the time.

Figure 4 shows a mashup model in which M_i denotes the mashup entities. Especially, we designate M_0 is the original content, and the remaining M_i are the mashlets. W_i is the DOM object `window` containing M_i. $E_{i,j}$ is the j_{th} element in M_i. In our design, all the mashlets are embedded in `<IFRAME>`, so that each mashlet has a separate `window`. Under the SOP, if $E_{2,1}$ wishes to access $E_{1,1}$ (when M_2 and M_1 are from different sources), the communication would fail. It guarantees that no unauthorized cross-domain access will succeed. Hence, we build a `postMessage` channel for the access.

When the user sends a mashup page request (i.e., original content) to the integrator, the proxy records the request and fetches the ACP files (in XML format) from the integrator and all the providers, and it automatically generates a site-specific ACP for this mashup. After the response is sent back to the proxy, the proxy inserts a link (that points to the generated ACP enforcement code written in JavaScript) in the HTML file and returns the modified HTML to the user. The enforcement code will be downloaded and enforced later by the browser. The procedure for requesting mashlet pages is similar. We describe the detail steps of the procedure in next subsections.

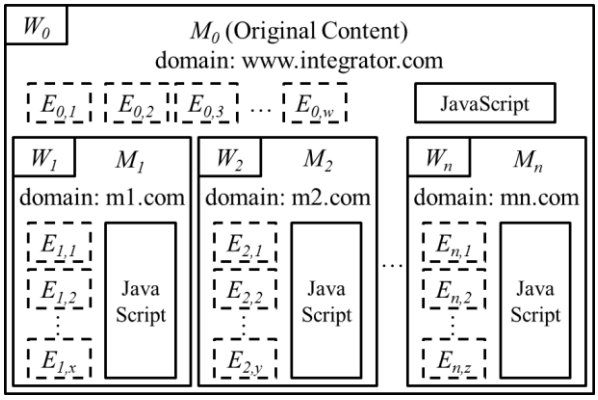

Figure 4 The web mashups model in the browser.

3.3 Site-Specific Access Control Policy

Figure 5 shows an example of the provider's ACP file and integrator's ACP file. `ACP_Provider` is the root node of the provider's ACP XML file, and its attribute `src` specifies the provider's URL. `Authorized_Element` (with an attribute `id`) lists the elements that the mashlet would like to share with others. The `Licensee` node specifies the authorized domain that can access this element. The integrator's ACP file is similar, but it has one more node, called `Provider_URLs`. `Provider_URLs` lists all the providers in this mashup. Our proposed proxy extracts this node to download providers' ACP files and generates the aggregated site-specific access control policy. The site-specific ACP is a table that specifies whether or not an element $E_{i,j}$ can be accessed by M_x. it is then used for generating the enforcement code.

According to the example in figure 5, we can conclude that the mashlet (www.craigslist.com) can access the element of `lcat_value` stored in the original content (www.housingmaps.com), and map mashlet (maps.google.com) can read the element of `craigslist_data` in the house mashlet (www.craigslist.com).

```
<ACP_Provider src="www.craigslist.com">
  <Authorized_Elements>
    <Authorized_Element id="craigslist_data">
      <Licensee> maps.google.com </Licensee>
    </Authorized_Element>
  </Authorized_Elements>
</ACP_Provider>
```

```
<ACP_Integrator source="www.housingmaps.com">
  <Provider_URLs>
    <Provider_URL>  maps.google.com </Provider_URL>
    <Provider_URL>  www.craigslist.com </Provider_URL>
  </Provider_URLs>
  <Authorized_Elements>
    <Authorized_Element id="lcat_value">
      <Licensee> www.craigslist.com </Licensee>
    </Authorized_Element>
  </Authorized_Elements>
</ACP_Integrator>
```

Figure 5 An example of ACP XML files.

3.4 The Operation of the Trusted Proxy

Based on the aggregated site-specific ACP, the proxy generates access control-related Java-Script files for each mashlet and the original content. Before the proxy returns the HTML pages to the browser, it inserts an HTML tag, named `<SCRIPT>`, in the `<HEAD>` of the HTML document to include the corresponding JavaScript file in the mashlet as well as in the original content. Hence, our proposed library and the access control script are added to the mashlets and the original content. Note that additional network overhead may occur when the proxy fetches the ACP XML files form the integrator and the providers. However, these ACP XML files can be cached at the proxy for better performance. We will discuss some performance issues in Section 5. Table 1 shows the operation functions supported by the proposed proxy.

Table 1 The operations of the proposed proxy

Operation	Description
Policy retrieval	The proxy retrieves ACP files from the providers and the integrator. These files are stored in a fixed directory of public available web servers, and are written by the providers and the integrator in a predefined XML format that is described in the previous section.
Access-control policies aggregation	According to all the related access control policies, the proxy generates an aggregated table for each web mashup. This operation restarts every time when any related policy has been updated.
Web scripts generation	According to each aggregated table, this task generates a JavaScript file that provides provider's mashlet and integrator's original content with APIs to use for secure and flexible client-side communication. Repeat this operation when the aggregated table has been updated.
Web page Adaptation	Insert a <script> in the head element of a HTML file to include a JavaScript library in every mashlet as well as the original content. Entities will call the function defined in the library.
Caching	To increase performance, in a reasonable period, processed results can be cached and used directly by the next request for the same URL. Furthermore, it can be done offline.

3.5 The Operation of the Browser

In the *all-trust* model (figure 6a), under the SOP, M_1 and M_2 can access any elements in W_0, because M_0, M_1 and M_2 are from the same source, i.e., the integrator. In this example, M_1 and M_2 are embedded in M_0's HTML document by `<DIV>`, so there is only one window (i.e., W_0) in this mashup. As mentioned, the elements of M_0, M_1, and M_2 can be accessed by a built-in JavaScript function call, for instance `document.getElementById()`. In the *no-trust* model (figure 6b), if M_0, M_1 and M_2 derive from different sources, and M_1 and M_2 are embedded in M_0 by `<IFRAME>` (i.e., M_0, M_1 and M_2 have their own `document` and `window`), a browser cannot exchange data under the SOP.

We use figure 6b for discussing HTML 5 `postMessage` method here. If M_1 can obtain the reference of M_2's window object, W_2, and apply the W_2.`postMessage()` method, cross-domain communication can be realized. However, M_1 may not have enough information about its sibling M_2, and cannot get W_2. Hence, in our design, our library will redirect the request of M_2 to M_0 (i.e., the parent of M_1 and M_2). According to the ACP, M_0 can explicitly allow or decline this redirected request for M_1. It makes the communication between mashlets easier.

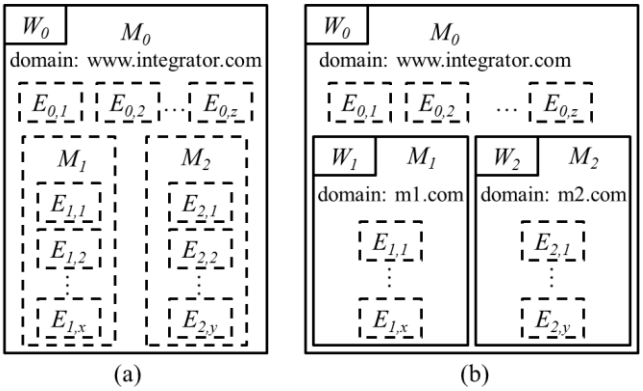

Figure 6 The legacy all-or-nothing trust models.

In our model, the mashlet only needs to know the domain and id of the element that it would like to communicate with. If M_1 wants to get M_2's element $E_{2,1}$, our library applies the following procedure for M_1 (see figure 7). (0) `entityRegisterHandler`: Every entity must add a handler to deal with the `postMessage` event. (1) `entityGetWindow`: M_1 asks its parent window, W_0, for the window index of element $E_{2,1}$. Our API secretly creates a secure `postMessage` channel between W_0 and W_1 for the query. Since the proxy has obtained the ACP files of M_0, M_1 and M_2, it has the structure of this mashup and already stores the index in W_0's JavaScript snippet. W_0 then returns the index of W_2 to M_1 by using `postMessage` channel as well. (2) `entityRequestData`: After obtaining the index of W_2, M_1 creates another `postMessage` channel to M_2. M_2 registers an event handler to catch the event if the element is designed to be accessible. (3) `querySendValid`: According to the aggregated site-specific ACP, M_2's JavaScript snippet contains information about the domains that can access the element $E_{2,1}$. The API returns `TRUE`, if M_1 passes the validation; otherwise a `NULL` object is returned. (4) `sendData`: M_2 then uses W_1 to create yet another `postMessage` channel to send the data to M_1. (5) `queryReceiveValid`: M_1 also needs to validate the data received from M_2 to prevent data from a malicious mashlet, and then M_1 gets the received data. Note that, the mashlet only needs to call one function to send a cross-domain message, since all the underlying communications between M_0, M_1 and M_2 are securely performed by our library automatically.

For this example in figure 7, there are four `postMessage` channels are established by our proposed library. Because the `postMessage` method only allows data to be "pushed" to the requester, the requester can only passively wait for data owner to send data. However, the mashlet developers do not need to deal with the complex communication procedures if our library is used.

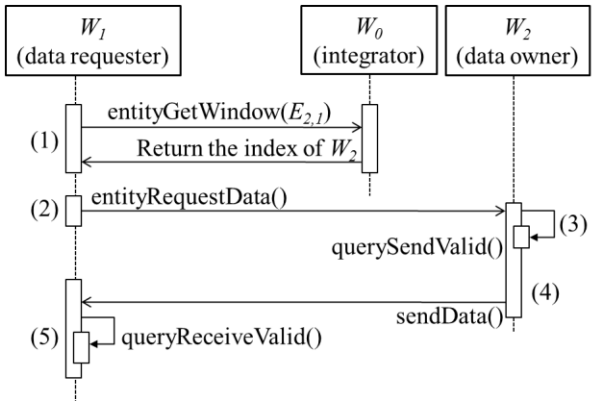

Figure 7 The cross-domain communication procedure in the browser.

3.6 The Pseudo Script of Cross-Domain Communication Library Used in the Browser

We show the pseudo scripts (shown in figure 8) of our proposed cross-domain communication library in this subsection. `Proxy.js` is the JavaScript file inserted into original HTML with the site-specific ACP by our proxy server. It provides secure cross-domain communication for the mashup developers. The entire `Proxy.js` has 382 lines of code. In the library, we create an object to handle the cross-domain procedure, and it extends `EventTarget.prototype` to add, save and launch event

listeners defined by each mashup entity. Each entity can define their customized actions in their `callbackFunc` functions, such as showing a message box that displays the data requested from others or storing the obtained data in their cookie for further use.

By calling `p_registerHandler`, we add an event listener, `p_onPostMessage`, to the current window to listen and handle the message sent by the original `postMessage` function. For Internet Explorer, the function named `attachEvent` for event registration is used; otherwise `addEventListener` is used for other the supported browsers. We also generate several hidden JavaScript text nodes using `document.createElement()` as temporary containers to allow `p_onPostMessage` to save received messages in the mashlet for later use.

`p_GetWindow(aURL)` is used to search the window objects embedded in the original content. The original content would help the caller to find the target window object having the specified domain or URL. The target window will be returned to the caller by another `postMessage` channel between the original content and the caller. Such design would be practical for the original content to find all embedded iframes, since a mashlet may not access its sibling mashlets due to the same origin policy.

`p_requestData(targetWin, targetURL, elementID)` is used by the caller to send a request to the callee after the caller gets the target window. If the `targetWin` is the window of the original content, the caller simply calls the `postMessage` function to send data by using its parent's window (because the original content is the mashlet's parent node). If the callee is a sibling mashlet, as we mentioned, `p_getWindow()` can obtain the window via the original content, so that our library can solve the problem of accessing sibling mashlet.

`p_onPostMessage` is responsible for handling the message sent between the entities (i.e. original content and mashlet). The message carrying in the event `e` has two parts: message type and message content. There are four message types are defined in our library: `REQUEST`, `REPLY`, `RETURN_WIN`, and `QUERY_WIN`. `REQUEST` and `REPLY` are used for data exchange between the entities. `RETURN_WIN` and `QUERY_WIN` are used for a mashlet and its original content to exchange window index of other entities. Once the `window` receives a message, `p_onPostMessage` will be invoked and perform the corresponding reactions.

`p_queryReceiveValid` is used by the data requester to check if it is secure to receive the client-side cross-domain data, while `p_querySendValid` is for the data owner to check if it is against its access control policy to send data to the requester. The site-specific ACPs generated by our proxy are used in these functions to check the validity of such cross-domain data delivery.

`p_sendData` is used by `p_onPostMessage` to perform different reactions corresponding to the message type. It is used by the data owner to send the data after receiving a `REQUEST` message.

`callbackFunc (event, args)` is the call back function of the event handler. Mashlet developers can manually register these self-defined functions in their mashlets. Once the `message` is received by the mashlet, the customized function will be invoked automatically.

```
p_registerHandler() {
        // Add an event listener to the current "window" for different browsers
        if (typeof window.addEventListener != 'undefined')
                window.addEventListener('message', p_onPostMessage, false);
        else if (typeof window.attachEvent != 'undefined')
                window.attachEvent('onmessage', p_onPostMessage);

        // Create some nodes in the "document" for saving certain information
        document.createElement(…);
}

p_getWindow(str,aURL){
        var iFrames = document.getElementsByTagName("iframe");
        for each iframe in iFrames
                var iframe_domain = iframes.getAttribute("src");
                return if find a matched aURL and iframe domain
}

p_requestData(targetWin,targetURL,elementID) {
        var msg = "REQUEST"+ elementID + nonce;
        if targetWin is the original content window
                window.parent.postMessage(msg, targetURL);
        else //target is a sibling mashlet
                top.frames[targetWin].postMessage(msg, targetURL);
}

p_onPostMessage(e) {
        var proxyObj = new Proxy();
        proxyObj.addListener("onNormal", callbackFunc, origin+"  "+e.data);

        // Phrase certain information carrying in the e.data,
        // such as message type, elementID, token, domain, URL, and etc.

        if (the massage type is REQUEST) {
                sendValid = p_querySendValid(elementID, e.origin);
                if(sendValid) p_sendData(sendValid, elementID, token, e); }
        else if (REPLY) {
                receiveValid = p_queryReceiveValid(e.origin, elementID);
                proxyObj.addListener("onReply", callbackFunc, msgSendToEntity);
                proxyObj.onReply(); }
        else if(RETURN_WIN) {
                proxyObj.addListener("onReturnWin", callbackFunc, query_result);
                proxyObj.onReturnWin(); }
        else if(QUERY_WIN) {
                query_result = p_getWindow(str, URL);
                p_returnQueriedWin(query_result, e); }
        else proxyObj.onNormal(); // perform original callback function
}

p_sendData(sendValid, elementID, token, e) {
        // if this sending action is valid, then send e using the real postMessage()
        if (sendValid) e.source.postMessage(REPLY_MSG, e.origin); }

p_querySendValid(elementID, requester) {
        Check p_ACPTable to see if the XML configuration file allow such request. }

callbackFunc (event, args){…}
```

Figure 8 The pseudo code of the cross-domain communication library.

4 System Implementation

4.1 Development Environment

We employ several servers in a campus network. The user and the proxy are in the same subnet, and each mashlet has a different domain. Our HTTP proxy is a Linux box (Fedora Core 5) with Apache 2.2 and Squid 3.1.1. It has a Pentium D 3.0 GHz CPU and 4G RAM. The web user has a PC having an Intel Core 2 Quad 2.33G CPU with 2G RAM. Microsoft Internet Explorer 8, Mozilla Firefox 3 and Google Chrome 6 along with the measurement tools (i.e., IE 8 Developer Tools, Firebug, and Chrome Web Developer Tools) are used.

4.2. The Squid Proxy and eCap Extension

We configured the client browser to connect with the Squid proxy. We adopted an extension of Squid, called eCAP [5], to perform content adaption. Figure 9 shows the operation of the proxy with the browser and web servers. Our modified eCAP adapter is about 800 lines in C++. The main entry point of our program is in function `adaptContent`. First, we insert our JavaScript library into the returned HTML files. Second, the aggregated ACP files and the JavaScript snippets is generated by another process, called *ACP Generator*, which is written in C++ (360 lines). We separate Squid/eCAP and the ACP Generator into two processes so that the tasks can be executed in parallel.

In figure 9, when the response of a web page from the integrator is returned to the proxy, Squid captures it and sends it to the eCAP. Our eCAP records its URL, passes it to the ACP Generator using shared memory technique, and inserts a JavaScript link into the HTML, e.g., `<SCRIPT SRC = "http://proxy_ip/integrator_M0.js">`. The modified HTML is sent back to the browser, which will then request the JavaScript code, `integrator_M0.js`, from the proxy. In the meantime, when a URL is sent to the ACP Generator, it forks a process to download the integrator's ACP XML file. Then, we produce the JavaScript code, i.e., `integrator_M0.js`, for the integrator. The code contains all the cross-domain communication APIs and the aggregated ACP for the integrator. As mentioned, the browser will download the code later after parsing the JavaScript link that we inserted into the HTML.

After parsing the integrator's ACP file and extracting the list of mashlets (from the `Provider_URLs` nodes), the ACP generator then further collects all the providers' ACP files and generates corresponding JavaScript snippet files for each provider, e.g., `provider_M1.js`. These JavaScript files will be downloaded later, when each mashlet we page is downloaded. We used Xerces-c++/3.1.1 and libcurl/4-7 to process the downloaded ACP files. Xerces-c++ is an XML parser for C++ which we use to process the ACP files. Libcurl is used to download ACPs from remote servers.

The procedure for retrieving a mashlet web page is much simpler. After the browser receives the original content, it sends extra HTTP requests to download the mashlet web pages. Squid also captures the requests and replies. At this point, eCAP does not need to retrieve the providers' ACP files, because they were obtained when we generated the aggregated ACP files. eCAP inserts a script link into the HTML reply, e.g., `<SCRIPT SRC = "http://proxy.com/provider_M1.js">`, and the JavaScript code for the mashlets will be downloaded later.

The pseudo code of the ACP Generator is in figure 10.

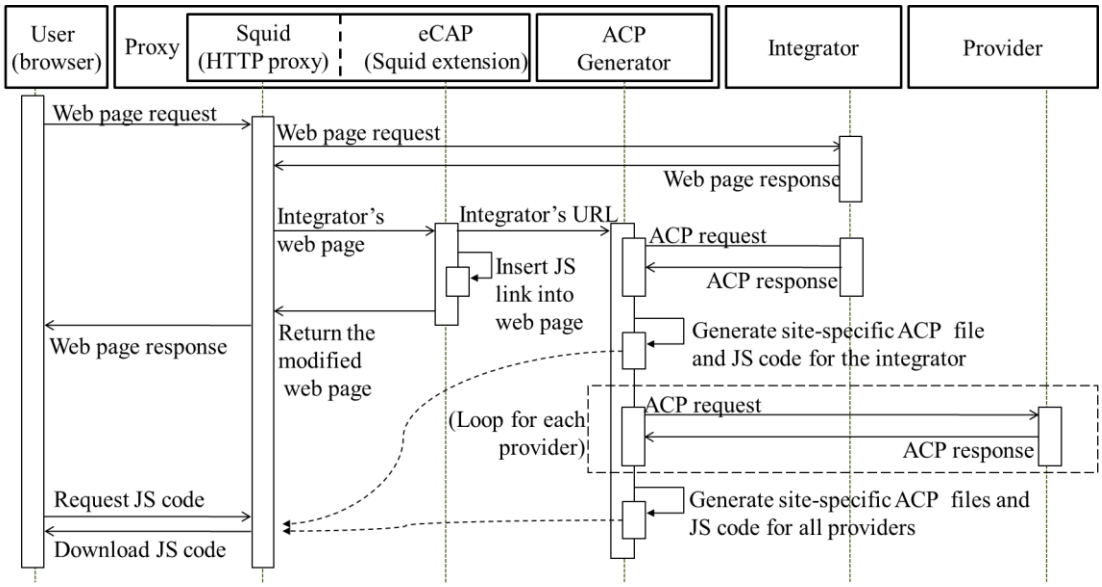

Figure 9 The procedure for cross-domain communication between browser, proxy (including Squid, eCap & ACP Generator), the integrator and a provider.

```
main()
{
    int shmid; // shared memory for eCAP and ACP Generator
    char *shm;

    CREATE_ATTACH_SHARED_MEM();
    while(0) // continue receiving the url sent by eCAP
    {
        string url = GET_URL_FROM_SHM();
        GET_ACP(url);
    }
}

GET_ACP(string url)
{
    char* integrator = DOWNLOAD_ACP(url);
    PARSE_XML(integrator);
    GENERATE_JS(integrator);

    for each provider in XML node <Provider_URL>
    {
        char* p = DOWNLOAD_ACP(provider);
        PARSE_XML(p);
        GENERATE_JS(p);
    }
}
```

Figure 10 The pseudo code of the ACP generator.

4.3 Ease of Use of the proxy.js Library

For mashlet developers, the cross-domain communication is done by our proposed library automatically. We set our entry point in an `init` function that will be called once the web page is fully loaded to the browser while the mashlet developers need not to manually include it. By calling our `p_registerHandler()` function (see section 3.6), the `init` function sets all the event handlers used in our library. Therefore, when a developer need to perform a cross-domain communication, we can use the proposed wrapper `p_requestData(targetWin, targetURL, elementID)` for the entities to request data from other entities and get window information `p_queryWindow` through original `postMessage` function.

Basically, the only thing that the mashlet developer should concern is the access control of the elements they would like to use and share. They only have to set the `tratgetURL` and `elementID` in the access control XML file specified in figure 5, and specify what `tratgetURL` and `elementID` to use in the runtime.

5 Evaluation

5.1 Security Analysis

Confidentiality. The cross-domain communication in our system is made by our library using the `postMessage` method. Our library guarantees that only the caller will get the `window` object. In this way, the confidentiality can be maintained, since no others can get the object under the restriction of the SOP.

Furthermore, our JavaScript library uses the URL of the callee as an input argument for the `postMessage` method. This argument is examined by our library to make sure it is the same as the source of the `window` object. If the examination fails, the event handler of `Message` will never be called. Our library also checks if the data requester is authorized to make the request to the owner. It prevents the violation of the web developer's ACP setting.

Integrity. The `postMessage` channel can be seen as a point-to-point channel. When a requester receives a message from the owner, no other entities can obtain the reference of the `Message` event object. In other words, no other entities can access the transmitted data in order to modify it.

Authenticity. Our API requests the caller and the callee to identify themselves (i.e., the domains they belong to). The domain is examined to ensure that it is the same as the source of the window object.

Our work relies on the firm Same Origin Policy to guarantee the securities. It highly depends on the browser's enforcement. Although like all other client-side scripts, a malicious user can always attack the client side data by viewing the source code download by the browser. However, in our work we point out that a malicious mashlet developer cannot access an unauthorized element via cross-domain communication mechanism when our access control mechanism is present. The enforcement of the Same Origin Policy and our proposed function wrappers will guarantee the communication securities as a whole.

5.2 Performance Analysis

In the subsection, we consider the overhead of our proxy-based approach. We assume that our system and a general proxy has similar network processing delay and queuing delay, but our proxy's processing delay is non-negligible. First, we analyze the time overhead for a general HTTP proxy environment and then compare it with that of our design.

1) The Latency of a General Proxy

Let BQ be the size of a request packet, BR be the size of a response packet, $R_{B\text{-}P}$ be the bandwidth between the browser and the local proxy, and R_I be the bandwidth between two Internet hosts. In addition, let P_Q be the time required by the proxy to process a request packet, and let P_R be the time for a proxy to handle a response. The total time needed by a general proxy to process a request and a response is as follows.

$$(B_Q/R_{B\text{-}P} + P_Q + B_Q/R_I) + (B_R/R_I + P_R + B_R/R_{B\text{-}P}) \quad (1)$$

Therefore, we expect that the total time needed for a web mashup containing one integrator and n mashlets will be

$$(1+n)*[(B_Q/R_{B\text{-}P} + P_Q + B_Q/R_I) + (B_R/R_I + P_R + B_R/R_{B\text{-}P})] \quad (2)$$

However, if the browser can send requests parallelly and the proxy can handle them simultaneously, the ideal factor of (2) will be *(1+1)*, not *(1+n)*. That means the n requests to the mashlets can be processed simultaneously.

2) The Latency of the Proposed Proxy with eCAP

Let *n* be the number of mashlets in a mashup, *m* be the number of trusted domains specified by a mashlet (or by an integrator), and *e* be the number of accessible elements in a mashlet (or in an integrator). In addition, let $B_{XML}(m, e)$ be the size (in bytes) of an ACP XML file for a mashlet (or an integrator) that has *m* trusted domains and *e* elements. And let $B_{ACP}(m, n+1, e)$ be the size of an aggregated ACP file for a mashup with *n* mashlets and one integrator that have *m* trusted domains and *e* elements. (In figure 5, *n* is also the number of `<Provider_URL>` nodes, *m* is the number of `<Licensee>` nodes, and *e* is the number of `<Authorized_Element>` nodes.)

Compared with (1), the extra time needed to retrieve the original content is as follows:

$$P'_Q + P'_R + B_{JS}(m,e)/R_I\,, \quad (3)$$

where P'_Q and P'_R are the extra processing times in our eCAP, and $B_{JS}(m, e)$ is the extra JavaScript code (in bytes) that must be downloaded for the cross-domain communication and enforcement of the ACP (see figure 9). In the JavaScript template used to generate JavaScript code, the value of $B_{JS}(m, e)$ is 11,026 bytes plus an ACP table. The fixed 11k JavaScript Code (about 400 lines) is our cross-domain communication library. The size of the ACP table is *(the average length of domains used) * m * (the average length of an element ID) * n* bytes. Figure 5 shows some examples of the domains and

element IDs. The length of the strings should not be too long in practice. (In our experiments, we assume they are both 16 bytes.)

The time needed to retrieve a mashlet web page is the same as (3). As mentioned earlier, for a mashlet, the eCAP adaptor does not need to trigger the operation of the ACP Generator.

The ACP Generator is designed to perform in parallel with the eCAP, as shown in figure 9. Its execution time can be divided into four parts: retrieving the integrator's ACP XML file, processing this file, retrieving the providers' ACP XML files, and processing them. They are formulated as follows:

$$\begin{aligned}&[B_Q/R_I + B_{ACP}(m, n+1, e)] + P_{ACP}(m, n+1, e)\\&+ n*[B_Q/R_I + B_{XML}(m, e)/R_I] + P_{XML}(m, e)\end{aligned} \tag{4}$$

The processing time of $B_{XML}(m, e)$, i.e., $P_{XML}(m, e)$, and that of $B_{ACP}(m, n+1, e)$, i.e., $P_{ACP}(m, n+1, e)$, will be measured with different sets of parameters in the next subsection. Moreover, similar to (2), the factor n of this equation would be 2 in an ideal case. In our system, the value of $B_{XML}(m, e)$ and $B_{ACP}(m, n+1, e)$ can be derived by analyzing the lengths of the XML tags and structure used in the figure 5. Based on our XML design, the $B_{ACP}(m, n+1, e)$ is [133 + 45 * n + e * (63 + 37 * m)] and the $B_{XML}(m, e)$ is [94 + e * (63 + 37 * m)].

3) Discussion

With regard to the retrieval time for web mashups, the main difference between a general proxy and our design is the extra processing time required by our eCAP (i.e., P'_Q and P'_R) and the extra time needed to download $B_{JS}(m, e)$. They represent the cost of secure cross-domain communication. We show the results of the overhead in our system in the next subsection.

ACP generator is another overhead. Although the generator can be run offline and in parallel, we still measure the amount of time required to generate ACP files. If the time is short enough (compared to the time spent on network transmission), due to the parallelism, we view the overhead as negligible.

If the browser supports sending simultaneous requests, the n factor in (2) and (4) can be set to 2. Unfortunately, in our experiments, the three browsers do not send requests in parallel. However, this issue depends on the browser's design and implementation, which is not the focus of our paper.

5.3 Performance Measurements

In the subsection, we present certain measurable evaluation results. For Internet Explorer, we use HTTP Watch Professional to measure the page load time in IE. It is an extension for Internet Explorer that gives page load timings and analysis of requests. For Firefox, Firebug 1.5.4 is used to measure the page load time. Firebug is a free extension for Firefox that gives page load timings and a number of development capabilities. As for Google Chrome, it has a built-in tool for performance measurement.

1) The Generation Time of ACP and JavaScript

In the ACP Generator, the major overhead results from generating ACP files and its JavaScript snippets, i.e., $P_{ACP}(m, n+1, e) + P_{XML}(m, e)$. We expect that the generation time will increase with the

number of domains (*m*), the number of mashlets (*n*), and the number of accessible elements (*e*). Figure 11 shows the generation time for different mashup settings in our experiments. From the results, we conclude that the time needed to generate ACP files in our system has a linear correlation with the numbers of *m*, *n*, and *e*.

Let us take the parameter setting used in our mashup environment (*m*, *n*+1, *e*) = (16, 2+1, 32) as an example. The ACP generator has to check 16*3*32 elements and generates the corresponding access control code for the mashup. In our system, the generation time for all ACPs and JavaScript snippets is 109.97 ms. $B_{ACP}(m, n+1, e)$ is 40,127 bytes and $B_{XML}(m, e)$ is 39,998 bytes. If R_I is 100 Mpbs, the time required to download an ACP XML file from one web server to the proxy is about 3.2 ms plus RTT. In today's network environment, the RTT might be a more significant concern. Hence, we suggest using the caching mechanism in the proxy to reduce this overhead

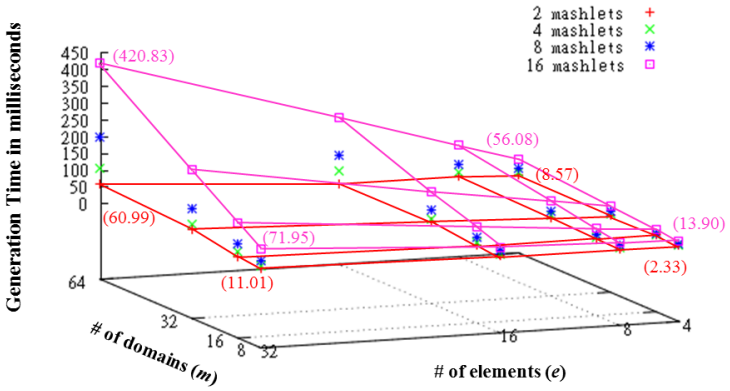

Figure 11 The ACP file generation time of different mashup settings.

2) The Time Required Manipulating the Returned HTML

In figure 9, we insert a script link in the HTML responses so that the browser knows where to download the cross-domain communication and ACP enforcement code. This task includes searching for an appropriate insertion point in the HTML document (usually in `<HEAD>` tag) and saving the record in the memory for later reference. On average, this task only takes 0.51 ms in our prototype system, which is reasonable.

3) Mashup Loading Time in Different Browsers

We measure the entire loading time of our cloned HousingMaps mashup at the browser. The parameter setting of our HousingMaps is (*m*, *n*+1, *e*) = (16, 2+1, 32). To ensure we get the correct measurement, we clean the browser and proxy caches every time we make a mashup request. Although the result is highly dependent on the parameter selected, the implementation of the browsers and the network delay, it gives us a high-level overview of system performance. We also measure the loading time of a mashup without the Squid proxy, as well as the loading time with the Squid but eCAP is disabled.

Figure 12 shows the average mashup loading times under the compared browsers. In Chrome 6, Firefox 3 and IE 8, the average overhead of our design (compared with that of the non-proxy

architecture) is about 743, 1049, and 1225 milliseconds, respectively. However, we notice that the overhead is caused primarily by the Squid proxy, not our eCAP adapter.

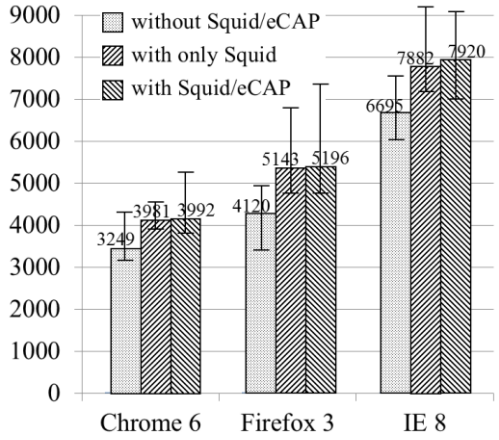

Figure 12 The loading time of different browsers.

4) The Time Required for Cross-Domain Communication in a Browser

Although the time required for cross-domain communication in the browser depends on the performance of the JavaScript engines, we still show the results of calling our library under the compared browsers (see figure 13). The average time needed to perform one fine-grained cross-domain communication in IE, Firefox and Chrome is 5.33, 0.22 and 0.08 milliseconds. The time required is nearly linear to the number of calls.

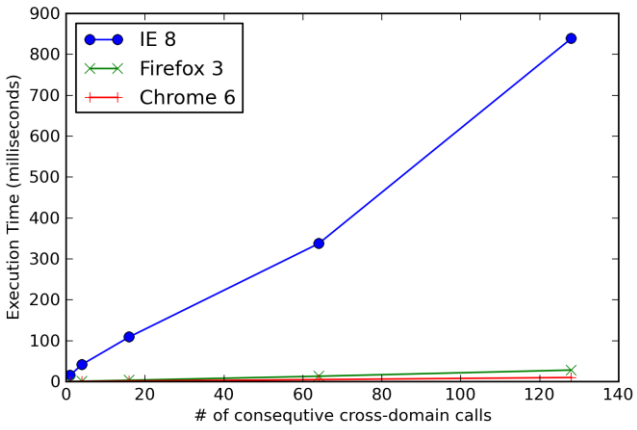

Figure 13 The execution time of different browsers.

6 Related Work

Fragment identifiers, the string after the # in a URI, can be used for cross-domain and cross-frame communication. Because the `location` property of an `<IFRAME>` can be modified by its parent window and itself, it can be used as a channel between the original content and the mashlet. However, fragment identifiers were not designed for use in this manner at the first place. As noted in [4], such ad hoc schemes require careful synchronization between the communicating parties, and it can be easily disrupted if the user presses the browser's back button.

In Subspace [4], the authors use domain promotion techniques to allow each provider to share a single JavaScript object with the integrator for communication. Under this scheme, a multi-level hierarchy of frames coordinates the `document.domain` property to communicate directly in JavaScript. Like most frame-based mashups, a policy for descendant frame navigation is required to prevent gadget hijacking. Using Sub-space correctly requires a significant amount of work on the part of the web developer, especially for complex mashups containing distrusted code from many different sources.

SMash [6] uses the concepts of publish-subscribe systems and creates an event hub abstraction that allows the mashup integrator to securely coordinate and manage content and information shared by multiple domains. They assume that the mashup integrator is trustworthy. The event hub implements the access policies that govern communications among domains. Barth [7] discovered that SMash is vulnerable to attacks that impersonate messages exchanged between components. However, Subspace and SMash are considered as ad hoc schemes rather than long-term solutions.

The MashupOS scheme [3] includes new primitives for isolating web content and ensuring secure communication. It proposes its own abstractions for missing trust levels in both access-controlled content and for unauthorized content. It adds new structures, e.g., `<SANDBOX>` and `<OPENSANDBOX>`, to HTML with variations on the same origin policies.

Crites *et al.* [8] proposed OMash, which adopts the trust relationships defined in MashupOS and only uses a single abstraction to express them. The authors argue that MashupOS still relies on the SOP for enforcement, so it suffers all of the SOP's vulnerabilities and pitfalls, including cross-site request forgery (CSRF), DNS rebinding and dynamic pharming, whereas OMash does not.

The `<MODULE>` tag proposed in [9] is similar to an `<IFRAME>` tag, but the module runs in an unprivileged security context without a principal, and the browser prevents the integrator from overlaying content on top of the module. A module groups DOM elements and scripts into an isolated environment; and socket-like communications are allowed between the inner module and the outer module. Unlike `postMessage`, the communication primitive used with the module tag is intentionally unauthenticated, so it does not identify the message sender.

Caja [10], Google's open source project, allows web applications of different trust domains to communicate directly with JavaScript function calls and reference passing. It translates scripts to an enforced JavaScript subset and only grants the modified scripts the privileges they require. It is possible to isolate scripts from each other and from the global execution environment, i.e., the browser window, to the degree needed. However, providers must write their components in Caja.

Facebook Markup Language (FBML) [11] is currently the most successful customized HTML and is used in Facebook Platform. FBML is a subset of the HTML with some elements removed and newly added tags. The FBML serves many roles in Facebook. For example, it is used in secure conditions on the user profiles, in small snippets on the news feed, and in full-page batches on canvas pages. The goal is to support a versatile tag set and thereby help developers target the different settings.

The BrowserShield framework [12] provides controlled cross-domain client-side communication by preprocessing the mashlet's JavaScript code to ensure that it can only perform actions specified in a set of guidelines. It transforms a code at runtime on the client side or as a one-time transformation on the integrator. Furthermore, it can prevent some denial-of-service attacks, e.g., navigating the parent frame to a new location or the appearance of an endless sequence of alert dialogs.

AdJail [13] dynamically creates a shadow page and passes the non-sensitive content to the shadow page to perform ads recommendation. It focuses on the confidentiality of the content for advertisement environment, rather than a general cross-domain communication.

OMOS [15] takes the advantage of creating temporary hidden `<IFRAME>` and fragment identifier messaging. Al-though it supports secure communication in browser environments, as noted in [4], we also believe such method is an ad hoc scheme.

We list certain features of different cross-domain communication approaches in Table 1. We notice that each approach has its own advantages that may not be replaced by other ones. It highly depends on the mashup architecture and the application environment used. Since there is a need to inspect the elements, insert code, and change the HTML, we anticipate that the selecting a trustworthy party to help the cross-domain communication is quite important. For our proxy design, a neutral third party (or a benign mashup service provider) can always fulfill this requirement, just like an Authentication Center used in encryption process.

	Server-Side Approach	Client-Side Approach	Proxy Approach
Code Insertion	Yes	Yes	Yes
HTML Modification	Manually by the Mashup programmer	Manually by the Mashlet programmer	Automatically by the Proxy
Browser Modification	No. postMessage used.	New tag or plug-in	No, postMessage used.
Policy Enforcement	SOP	Plug-in	SOP + ACP
Main Overhead	Web server	Browser	Proxy
Browser Download		Plug-in (1 time)	
Policy Modification	Modify server policy	Modify HTML	Modify XML
Trust	Server	Plug-in	Proxy

Table 1 The comparison of different cross-domain communication approaches

7 Discussion and Future Work

7.1 Discussion of Related Work

According to [4], some browsers are now trying to restrict the use of ad hoc tricks to perform cross-domain communication. However, the solution may require adding new elements to the HTML

standard, browser changes (i.e., installation of a browser plug-ins or browser updates), and all web site developers involved to rewrite their codes. First, changing browsers requires users to update browsers or install additional plug-ins. For users who do not wish to perform software updates, their securities have been exposed to potential threats. Although browser plug-ins 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 [4].

Moreover, in [14], the authors point out some unsafe browser features that should be removed. Hence, ad hoc solutions might not be still workable in the future. We anticipate that solutions built on the top of standard `postMessage` method would be more appropriate. However, some future works still needs to be done, such as design of a trust model in Ajax environment and ACP caching mechanisms.

Other than using current standards, some working groups try to embed the cross-origin resource sharing mechanism into new HTML standard. Cross-Origin Resource Sharing (CORS) is such a specification that enables an open access across domain-boundaries. W3C makes a draft [16] that defines a mechanism to enable client-side cross-origin requests. It allows a server to reply a response that includes `Access-Control-Allow-Origin` herder, with the origin of where the request originated from as the value, to access certain resource's content. This specification extends this model in several ways to retrieve resources. The `Access-Control-Allow-Origin` header indicates whether a resource can be shared based by returning the value of the `Origin` request header in the response. It also specified how long the results of a request can be cached using the `Access-Control-Max-Age` header. The `Access-Control-Allow-Methods` header indicates which methods can be used during the request.

It takes time for the web server, browser vendors, as well as, the mashlet developers to support such new standard; after that, a common web user can enjoy the benefit of it. We anticipate that well-designed standards can improve the security of cross-domain communication in the near future. However, our proxy-style approach is still a valuable reference model for the mashp platform provider and standard developers.

7.2 Potential Risk of Introducing a Proxy

The proposed proxy is used to generate code snippets and insert these returned HTML documents to enforce the access control policies. We adopt the proxy-approach to avoid adding computation loading on the client (i.e., browser). However, the same functionality can be provided by a browser plug-in. They are essentially the same in a way that they both require the client to trust them to perform the access control without malicious intention. Once they are configured, they are given the right to control the privileges of every mashlets.

Their difference lies in the number of network requests and the network routing. However, proxy-based solution is suitable for enterprise network and it is easy to maintain the code without bothering the users. We can also perform some authorization procedures before the proxy configuration to assure the identity of the proxy, in case a user connects to a distrusted proxy.

7.3 Discussion of Proxy-Based Design

Our proposed proxy-based design could be used in an enterprise mashup model for business benefits. We anticipate that the proxy-based design can help the mashup platform providers to provide a better solution to the mashup developers and users. The access control mechanism can be a part of social network platform as a default element sharing mechanism in the mashup. Therefore, all the developers can benefit from the proxy design and do not need to spend time for sharing mechanism. A for-profit mashup platform provider can even integrate our mechanism in their customized web server to act as our proxy.

The proxy currently that we use is a general HTTP proxy; however, we believe it can be integrated in a browser as a plug-in to handle the cross-domain communication. However, it makes a general user to download and install it before using the mashup website, which is not that convenient. The most important of the proxy design concept is separating the access control and enforcement code from the original web content. We believe this concept is important to guarantee the security.

In addition, for a thin web client, such as mobile devices or battery-bounded devices, the proxy-based design can save the computation time and data transmission time of performing access control mechanism. The thin client does not need to compute and download the access control files directly from the servers; rather the proxy does so for the thin clients. They are all the advantages of using proxy-based design.

8 Conclusions

In this paper, we propose a proxy-based design for a mashup environment. Our scheme provides a guaranteed security framework that allows mashlets to perform cross-domain communication. The authenticity, integrity, confidentiality and flexibility requirements are addressed by the design. We build up a client-side cross-domain communication library on the top of the HTML 5 `postMessage` method with a proxy-style fashion. We implement a prototype system and test the overhead of the proxy. The results show the overhead is linear to the number of shared components, and the incurred overhead is reasonable in our experiments.

References

1. HousingMaps. http://www.housingmaps.com/
2. Ruderman, J. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2001, (accessed Aug 10, 2008).
3. Howell, J., Jackson, C., Wang, H. J. and Fan, X., MashupOS: Operating System Abstractions for Client Mashups. in Proceedings of 11th Workshop on Hot Topics in Operating Systems, (San Diego, CA, 2007).
4. Jackson, C. and Wang, H. J., Subspace: Secure Cross-Domain Communication for Web Mashups. in Proceedings of the 16th International World Wide Web Conference,(Banff, Alberta, Canada, 2007).
5. eCap. http:// wiki.squid-cache.org/Features/eCAP
6. Keukelaere, F. D., Bhola, S., Steiner, M., Chari, S. and Yoshihama, S., SMash: Secure Cross-Domain Mashups on Unmodified Browsers. Tech. Rep., IBM Research, Tokyo Research Laboratory, 2007.

7. Barth, A., Jackson, C., Mitchell, J. C., Securing Frame Communication in Browsers. Communications of the ACM, 52(6). 83-91. 2009.
8. Crites, S., Hsu, F. and Chen, H., OMash: Enabling Secure Web Mashups via Object Abstractions. in Proceedings of 15th ACM Conference on Computer and Communications Security, (Alexandria, VA, 2008).
9. Crockford, D. The Module Tag: A Proposed Solution to the Mashup Security Problem. http://www.json.org/module.html/
10. Miller, M. S., Samuel, M., Laurie, B., Awad, I. and Stay, M., Caja: Safe Active Content in Sanitized JavaScript. Google research project, 2008.
11. Facebook Markup Language (FBML). http://developers.facebook.com /docs/reference/fbml/
12. Reis, C., Dunagan, J., Wang, H. J., Dubrovsky O. and Esmeir, S., BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. in Proceedings of the 7th Symposium on Operating Systems Design and Implementation, (Seattle, WA, 2006).
13. Louw, M. T., Ganesh, K. T. and Venkatakrishnan, V. N., AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. in Proceedings of the 19th USENIX Security Symposium, (Washington, DC, 2010).
14. Singh, K., Moshchuk, A., Wang, H. J. and Lee, W., On the Incoherencies in Web Browser Access Control Policies. in Proceedings of the 31st IEEE Symposium on Security and Privacy, (Oakland, CA, 2010).
15. Zarandioon, S., Yao, D. and Ganapathy, V., OMOS: A Framework for Secure Communication in Mashup Applications. in Proceedings of the Annual Computer Security Applications Conference, (Anaheim CA, 2008).
16. Cross-Origin Resource Sharing. W3C Working Draft. http://www.w3.org/TR/cors/