A Secure Proxy-Based Cross-Domain Communication for Web Mashups

Conference Paper *in* Journal of Web Engineering (JWE) · October 2011

DOI: 10.1109/ECOWS.2011.10 · Source: IEEE Xplore

Shun-Wen Hsiao
National Chengchi University
20 PUBLICATIONS **101** CITATIONS
SEE PROFILE

Yeali Sun
National Taiwan University
105 PUBLICATIONS **1,553** CITATIONS
SEE PROFILE

Fu-Chi Ao
Carnegie Mellon University
1 PUBLICATION **9** CITATIONS
SEE PROFILE

Meng Chang Chen
Academia Sinica
85 PUBLICATIONS **962** CITATIONS
SEE PROFILE

A Secure Proxy-Based Cross-Domain Communication for Web Mashups

Shun-Wen Hsiao, Yeali S. Sun, Fu-Chi Ao
Department of Information Management
National Taiwan University
Taipei, Taiwan
{r93011, sunny, b93705043}@im.ntu.edu.tw

Meng Chang Chen
Institute of Information Science
Academia Sinica
Taipei, Taiwan
mcc@iis.sinica.edu.tw

Abstract—**A web mashup is a web application that integrates content from heterogeneous sources to provide users with a more 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 (SOP) implemented by the browsers cannot provide a flexible client-side communication mechanism to exchange information between different sources. To address this problem, we propose a secure client-side cross-domain communication model 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.**

Web Security; access control; mashups, same origin policy

I. 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 into one web mashups to enrich the web applications.

In *interactive mashups*, the content of one source may be able to communicate with the content of the other sources. HousingMaps [1] is an example of this type of mashup. It combines the property database of Craigslist with map data from Google Maps 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 [17]. 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 rendered.

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 such access rights. However, such inflexible policy may limit the development of interactive mashups, so the legacy SOP forces web developers to make tradeoffs 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 changes the source (e.g., domain) of the documents so that these documents can communicate with each other under the SOP.

For mashups to work effectively, we believe there should be a flexible and fine-grained access control model. The model should satisfy the following principles of security. (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 untrusted 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 (in 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 test 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 and gadgets, 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.

II. BACKGROUND

A. Web Mashups

There are several roles in web mashups. An *integrator* is a site that hosts web mashups; a *provider* is a site that provides content or service to web mashups; and a *mashlet* is client-side content from the provider, and 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*.

Fig. 1 shows an example of mashups with one original content (integrator: www.housingmaps.com), one map mashlet (provider: maps.google.com/maps), and one housing mashlet (provider: www.craigslist.com). 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 and interact with the user. If a mashlet is loaded by a <DIV>, then the original contain can access the mashlet without any constraint. If a mashlet is loaded by an <IFRAME>, the mashlet is isolated as a separate HTML document so that the access policies between the original content, the mashlet itself and other mashlets are specified and enforced by the SOP (See II.C).

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 loaded by <IFRAME>) is loaded into a DOM container, called a *window*. DOM is the standard platform-independent and language-neutral programming interface used by client-side scripts (e.g., JavaScript) to access and manipulate the content, structure and style of documents. The issue of the mashups that we address is how to design a flexible and fine-grained access control mechanism between these HTML documents from the heterogeneous sources.

B. Mashup Architectures

Server-side mashups (Fig. 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 integrator. Yahoo Pipes (http://pipes.yahoo.com/pipes/) utilizes this design.

Since the integrator is responsible for collecting data, the user may need to delegate authority to the integrator to obtain content from the providers. Such approach may have privacy and security concerns. Other drawbacks of server-side mashups include additional processing delays and limited scalability.

Client-side mashups (Fig. 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, it is necessary to consider the problem of cross-domain communication, since the original content and the mashlets usually come from different sources.

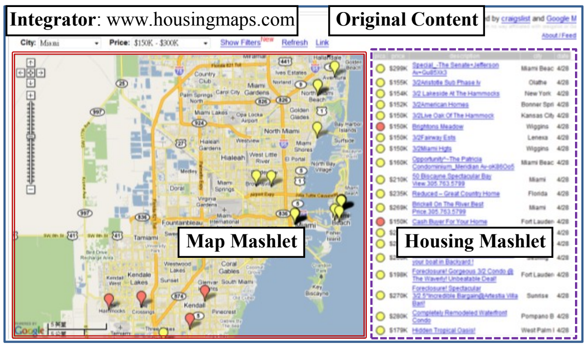


Figure 1. A web mashup example: HousingMaps.com.

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 (Fig. 2c). The proxy can generate a site-specific access control policy (ACP) and its corresponding client-side JavaScript snippets based on the mashlets' security settings.

C. Same Origin Policy

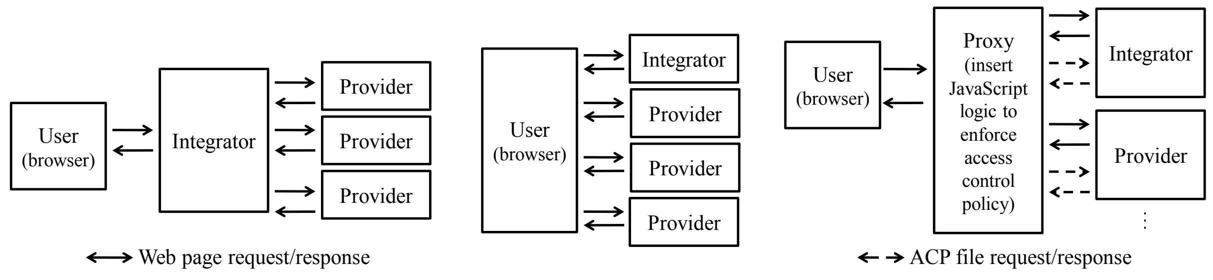
The Same Origin Policy (SOP) is enforced by the browser. It is assumed that two web pages derive from the same source (or origin) if the application layer protocol, port and domain are the same for both pages. Recall that the SOP is an all-or-nothing mechanism. For a server-side mashup, all the contents are from the "same source", i.e., the integrator, so there is a full-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. Contents from a different source run in an isolated environment with no access to other content.

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 tradeoffs between security and functionality.

D. HTML 5 postMessage

HTML 5 specifies an API, named postMessage, for asynchronous communication between DOM windows. To send a message, a window 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.

It seems convenient to perform cross-domain communication via the postMessage, however in practice there are some issues that need to be addressed. First, the postMessage receiver needs to validate the message on its own to ensure it is



(a) Server-Side Mashup **(b) Client-Side Mashup** **(c) Proxy-Based Mashup**

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

from an authorized source (domain). The mashlet programmer needs to write a long 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 efficient. Third, basically a mashlet has no information about the mashup and its sibling mashlets. However, the message sender needs to get the receiver's window id to invoke the postMessage method. In practice, it is difficult and not good for the sender to search for this information in the 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.

In our proposed system, we take the advantage of the postMessage, and also we provide a proxy-based solution to mitigate the issues mentioned above to allow fine-grained, efficient and secure interactions between mashup components.

III. System Design

A. Security 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. Our design must ensure that the mashlet can only see the proper content.

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. Also, it should ensure that the communication parties comply with the ACP set by the mashup developers.

Flexibility. We believe the ACP should be more fine-grained to the element level, rather than domain. 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.

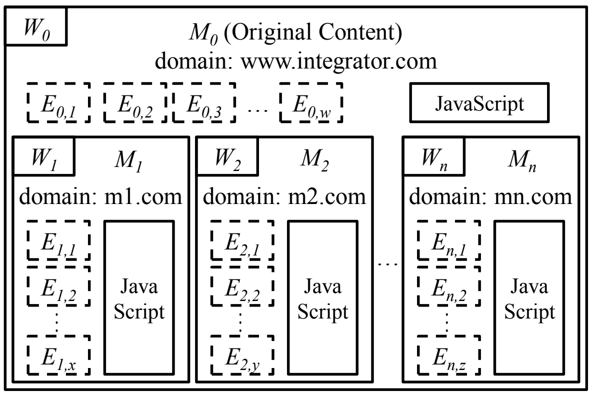


Figure 3. The web mashups model in the browser.

B. The Proposed Web Mashup Model

Fig. 3 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 DOM window. Under the SOP, if $E_{2,1}$ wishes to access $E_{1,1}$ (when M_2's and M_1 are from different sources), the communication would fail. It guarantees that no unauthorized cross-frame access will succeed. We will build another postMessage channel for the authorized 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 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.

```
<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 4. An example of ACP XML files.

C. *Site-Specific Access Control Policy*

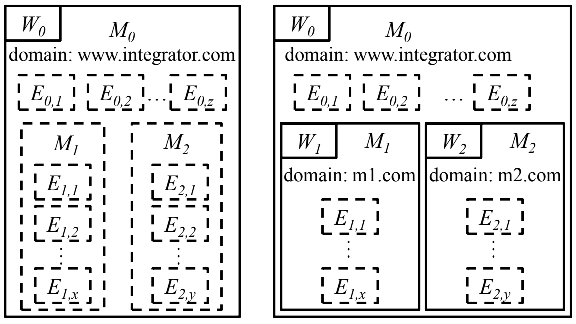
Fig. 4 shows an example of the provider's ACP file and integrator's ACP file. ACP_Provider is the root node of the provider's ACP XML file, and its attribute src specifies the provider's URL. Authorized_Element (with an attribute id) lists the elements that the mashlet would like to share with others. 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. A proxy can extract this node to download providers' ACP files and generate the aggregated site-specific access control policy. It is a table that specifies whether or not an element $E_{i,j}$ can be accessed by M_x. The site-specific access control policy is then used for generating the enforcement code.

D. *Operation of the Trusted Proxy*

Based on the aggregated ACP, the proxy generates JavaScript files containing our APIs for mashlets and original content. Before the proxy returns the HTML pages to the browser, it inserts an HTML tag, <SCRIPT>, in the <HEAD> of the HTML document to include the corresponding JavaScript file in the mashlet as well as the original content. Additional network overhead occurs when the proxy fetches the ACP XML files. However, the ACP XML files can be cached at the proxy for better performance. We will discuss some performance issues in Section V.

E. *The Operation of the Browser*

In the all-trust model (Fig. 5a), 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. M_1 and M_2 are usually embedded in M_0's HTML document by <DIV>, so there is only one window (i.e., W_0) in this mashup. The elements of M_0, M_1, and M_2 can be accessed by a built-in JavaScript function call, document.getElementById(), as long as the element's id is unique in the HTML document. In the no-trust model (Fig. 5b), 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>, a browser cannot exchange data across the window boundary because of the SOP.



(a) All-trust model. (b) No-trust model.

Figure 5. The legacy all-or-nothing trust models.

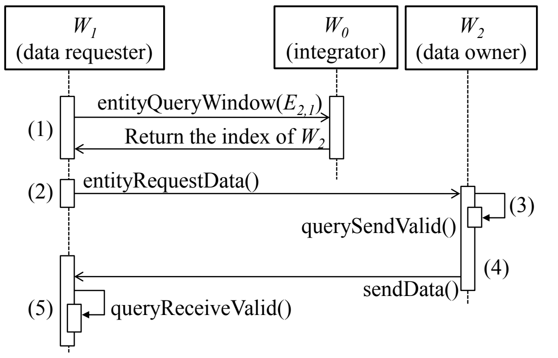


Figure 6. The cross-domain communication procedure in the browser.

HTML 5 provides the new postMessage method to cross the boundary. 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. Since M_1 may not have enough information about M_2, in our design, we make M_1 issue a request of M_2 to M_0. According to the ACP, M_0 can explicitly allow or decline this request.

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 (see Fig. 6). (1) entityGetWindow: M_1 asks its parent window, W_0, for the window index of $E_{2,1}$. 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 understands the structure of this mashup and already stores the index in W_0's JavaScript snippet. (2) entityRequestData: After obtaining the index of W_2, M_1 creates a 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 ACP, M_2's JavaScript snippet contains information about the domain that can access the $E_{2,1}$. The API returns TRUE, if M_1 passes the validation; otherwise a NULL is returned. (4) sendData: M_2 then uses W_1 to return the data via another postMessage channel. (5) queryReceiveValid: M_1 also needs to validate the data received from M_2 to prevent data from a malicious mashlet. M_1 then get 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.

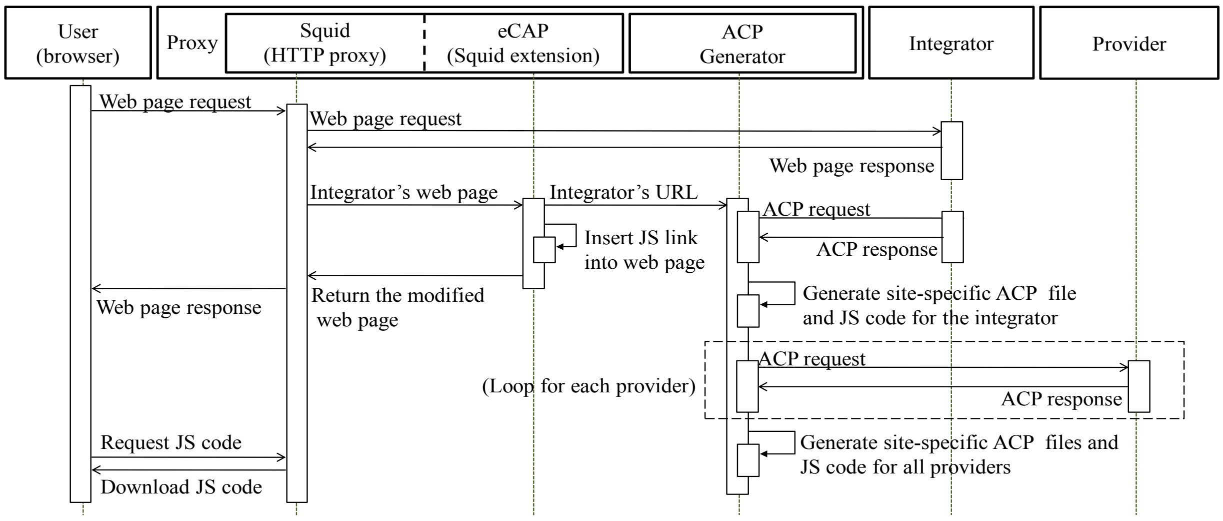


Figure 7. The procedure for cross-domain communication between browser, proxy (including Squid, eCap & ACP Generator), integrator and provider.

IV. System Implementation

A. 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 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.

B. The Squid Proxy and eCAP

We configured the client browser to connect with the Squid proxy. We adopted an extension of Squid, called eCAP [5], to perform content adaption. Our modified eCAP adapter is about 800 lines in C++. Fig. 7 shows the operation of the proxy with the browser and web servers. 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 Fig. 7, 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, and inserts a JavaScript link into the HTML, e.g., <SCRIPT SRC = "http://proxy.com/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.

The procedure for retrieving a mashlet web page is much simpler. (Due to page limit, it is not shown in Fig. 7.) After the browser receives the original content, it sends more 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 only 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.

V. Evaluation

A. 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. 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.

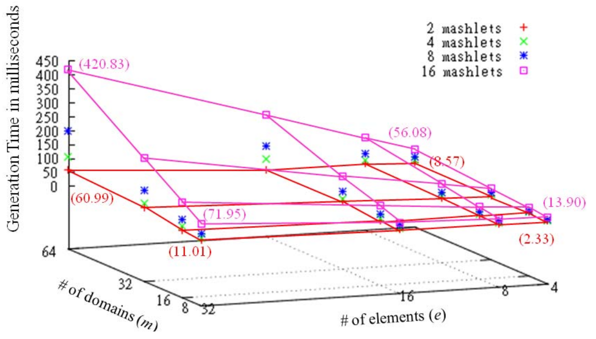


Figure 8. The ACP file generation time of different mashup settings.

B. *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) *A General Proxy*

Let B_Q be the size of a request packet, B_R 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.

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

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

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

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) *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 Fig. 4, n is 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:

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

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 Fig. 7). 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. Fig. 4 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). But 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 Fig. 7. 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:

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

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 in 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 III.C. 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.

C. *Performance Measurements*

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). Fig. 8 shows the generation time for different mashup settings in our experiments. From the results, we conclude that the time

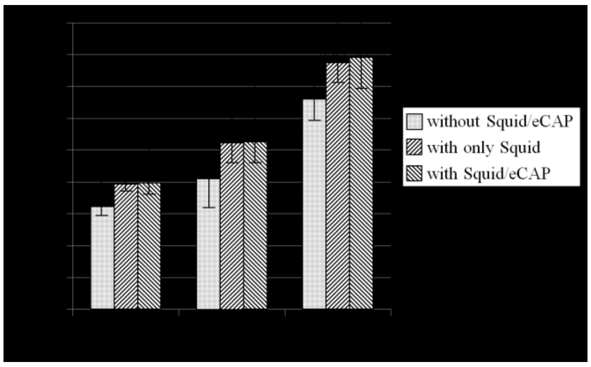


Figure 9. The loading times of different browsers.

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. In our system, the generation time for all ACPs and JavaScript snippets is 109.97ms. $B_{ACP}(m, n+1, e)$ is 40,127 bytes and $B_{XML}(m, e)$ is 39,998 bytes. If R_I is 100Mpbs, the time required to download an ACP XML file from one web server to the proxy is about 3.2ms 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.

2) The Time Required to Insert Script Link

In Fig. 7, 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.

Fig. 9 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.

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 Fig. 10). The average time

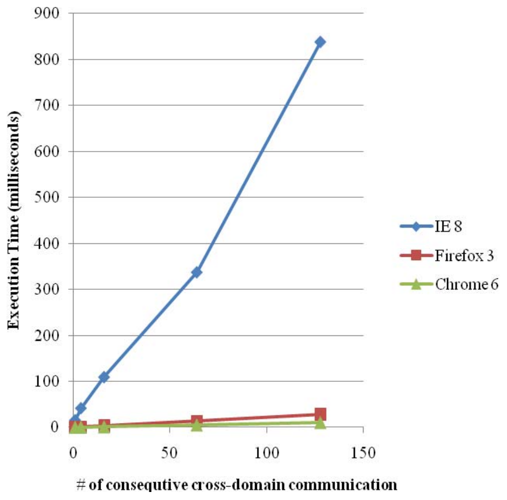


Figure 10. The execution time of different browsers.

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.

VI. Related Works

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 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 descendant frame navigation policy is required to prevent gadget hijacking. Using Subspace correctly requires a significant amount of work on the part of the web developer, especially for complex mashups containing untrusted 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 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. 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 is possible to translate scripts to an enforced JavaScript subset and to only grant the 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.

According to [4], some browsers are now trying to restrict the use of ad hoc tricks to perform cross-domain communication. Moreover, in [14], the authors point out some unsafe browser features that should be removed. We anticipate that solutions built on the top of standard postMessage method would be more appropriate. However, there are still some future works that needs to be done, such as design of a trust model in Ajax environment and ACP caching mechanisms.

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

VII. 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 shows 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] J. Ruderman. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2001, (accessed Aug 10, 2008).

[3] J. Howell, C. Jackson, H. J. Wang, and X. Fan, "MashupOS: Operating System Abstractions for Client Mashups," in *Proc. 11th Workshop on Hot Topics in Operating Systems (HotOS XI)*, May 2007.

[4] C. Jackson and H. J. Wang, "Subspace: Secure Cross-Domain Communication for Web Mashups," in *Proc. 16th International World Wide Web Conference (WWW)*. May 8-12, 2007.

[5] eCAP. http://wiki.squid-cache.org/Features/eCAP

[6] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama, "SMash: Secure Cross-Domain Mashups on Unmodified Browsers," Tech. Rep., IBM Research, Tokyo Research Laboratory, June 2007.

[7] A. Barth, C. Jackson, and J.C. Mitchell, "Securing Frame Communication in Browsers," *Communications of the ACM*, vol. 52, no. 6, Jun. 2009, pp. 83-91.

[8] S. Crites, F. Hsu, and H. Chen, "OMash: Enabling Secure Web Mashups via Object Abstractions," in *Proc. 15th ACM Conference on Computer and Communications Security*, Alexandria, VA, Oct. 2008, pp. 99-108.

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

[10] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe Active Content in Sanitized JavaScript. A Google research project, 2008.

[11] Facebook Markup Language (FBML). http://developers.facebook.com/docs/reference/fbml/

[12] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML," in *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[13] M. T. Louw, K. T. Ganesh, and V.N. Venkatakrishnan, "AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements," in *Proc. 19th USENIX Security Symposium (Security '10)*, Washington, DC, August 2010.

[14] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the Incoherencies in Web Browser Access Control Policies," in *Proc. 31st IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[15] S. Zarandioon, D. Yao, and V. Ganapathy. "OMOS: A Framework for Secure Communication in Mashup Applications," in Proc. *Annual Computer Security Applications Conference (ACSAC)*, 2008.