Privacy-Preserving Browser-Side Scripting With *BFlow*

Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris

Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory

Abstract

Some web sites provide interactive extensions using browser scripts, often without inspecting the scripts to verify that they are benign and bug-free. Others handle users' confidential data and display it via the browser. Such new features contribute to the power of online services, but their combination would allow attackers to steal confidential data. This paper presents BFlow, a security system that uses information flow control to allow the combination while preventing attacks on data confidentiality.

BFlow allows untrusted JavaScript to compute with, render, and store confidential data, while preventing leaks of that data. BFlow tracks confidential data as it flows within the browser, between scripts on a page and between scripts and web servers. Using these observations and assistance from participating web servers, BFlow prevents scripts that have seen confidential data from leaking it, all without disrupting the JavaScript communication techniques used in complex web pages. To achieve these ends, BFlow augments browsers with a new "protection zone" abstraction.

We have implemented a BFlow browser reference monitor and server support. To evaluate BFlow's confidentiality protection and flexibility, we have built a BFlow-protected blog that supports Blogger's third party JavaScript extensions. BFlow is compatible with every legitimate Blogger extension that we have found, yet it prevents malicious extensions from leaking confidential data.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection; H.3.5 [*Online Information Services*]: Web-based services

General Terms Design, Security

Keywords information flow control, labels, web platforms, JavaScript

1. Introduction

Three important trends in Internet-based computing have emerged in recent years. First, web sites are increasingly hosting sensitive user data and applications; hosted e-mail has been joined by hosted spreadsheets, confidential blogs, etc. Second, large swathes of web user interface code now run in the browser, as JavaScript and other browser scripting languages. Third, many web sites use JavaScript that they might not fully understand, including large imported libraries and even extension scripts written by arbitrary third party programmers. These extensions can use server-side APIs to access and manipulate users' server-based data, giving rise to application-like third-party extensions on "platform" sites such as Facebook [Facebook 2009] and Blogger [Blogger 2009].

The combination of third-party browser scripts and sensitive user data raises the possibility of scripts stealing confidential data. For this reason, today's web applications that value user privacy must forbid browser script extensions, or refuse to reveal sensitive user data to extensions. These approaches cut off useful behavior, undermining the value of extensibility. For example, web applications like Gmail would benefit from third party JavaScript extensions, but confidentiality problems make them difficult to support. As a substitute, Gmail users modify their *browsers* to do things like optimize Gmail's UI for particular mobile devices and alter the way Gmail renders email [Firefox 2009].[1]

Existing web sites that support extensions tend to do so with less sensitive, but still confidential data. For example, the Blogger web site hosts confidential blogs, yet permits users to install third-party JavaScript extensions, that they might not fully understand, on their blogs. These extensions can read confidential data, compute on it, and display it to the user (which is reasonable by itself), but they can also communicate any information they read to outside parties (which can violate the user's privacy). Part of the underlying problem is that the browser security policy gives all scripts that come from a given web site full privileges with respect to that site.

[1] The ideas in this paper may also apply to software-as-a-service sites, though we have not pursued this idea.

Recent work [Wang 2007, Miller 2008, Jim 2007] proposes improvements to today's browser security policy such as finer-grained separation of privileges between different parts of the browser. But these solutions still force users or developers to make up-front decisions as to whether or not to trust third-party code with confidential data. Mistakenly deciding "no" inhibits extensibility; mistakenly deciding "yes" invites data theft.

This paper describes BFlow, a new browser security system. BFlow lets browser scripts compute with confidential data while restricting their ability to reveal that data. BFlow uses a reference monitor in the browser to enforce information flow control (IFC), observing the communication of each script with other scripts and with web sites. These observations help BFlow decide whether each script has seen confidential data (whether directly or transitively through another script) and from what site that data came. The BFlow reference monitor uses the tracking information to restrict how data is revealed: if a script has seen confidential data, it can only communicate with the site whence the confidential data came unless that site explicitly permits communication with other servers. BFlow places few new restrictions on scripts that have not been exposed to confidential data. To take advantage of BFlow, a web site must cooperate by marking outgoing confidential data with security metadata and recording the confidentiality of incoming data.

The challenges in designing BFlow differ from those solved by operating system IFC systems [Bell 1976, McIlroy 1992, Dep 1985, Efstathopoulos 2005] because the browser has somewhat unusual notions of the principals that own data (web sites), of the natural code unit at which to apply IFC (the frame), and of the special flows of information that must be supported (among frames and to web servers).

We have implemented a prototype BFlow browser reference monitor as a Firefox plug-in. We have also implemented the server part of BFlow as a gateway layer that sits between an Apache web server and the web site's application logic. These implementations are intended to be easy to deploy: the Firefox plug-in is easy to install, and the BFlow reference monitor supports the full JavaScript language so that most scripts run with no changes.

To evaluate BFlow's privacy protection and flexibility, we implemented two web sites that incorporate third party JavaScript: a blog compatible with Blogger's third party extensions, and a social networking site that implements common application features in untrusted JavaScript. The blog example shows that many existing scripts will work with few modifications and that malicious JavaScript that leaks confidential data in Blogger does not leak within BFlow. The social network example shows that BFlow supports a wide range of third party functionality.

The contributions of this work are 1) a new technique that (when properly used) prevents JavaScript from stealing confidential data, and for example prevents all "cross-site scripting" attacks whose aim is to steal data; 2) a new information flow control model for the browser that is compatible with JavaScript's runtime and communication environment, and 3) an easy-to-deploy implementation.

2. Background: JavaScript

Web sites use in-browser JavaScript to provide high-quality user interfaces. This section briefly reviews what JavaScript can do within a browser, focusing on communication.

A browser consists of one or more *frames*, each containing a separate HTML document and JavaScript interpreter. Browser frames can contain sub-frames using the `frame` and `iframe` HTML directives. Each browser window or tab is a *top-level* frame, each frame that embeds a sub-frame is a *parent*, and each sub-frame is the *child* of its parent.

The browser represents the displayed document in each frame as a data structure called the Document Object Model (DOM). JavaScript code is allowed to read and modify the DOM of any frame from the same origin server as the code.[2] Different JavaScript code from the same origin can communicate via modification to each other's DOMs, and JavaScript can cause communication with any web server by modifying the DOM to fetch a page or image from that server.

The restriction that JavaScript only access DOMs from the same origin is called the *same-origin policy* (SOP). The SOP also only allows a script to send AJAX requests to its origin server. The high-level goal of the SOP is to guard the operation of each web site and its JavaScript from interference by other sites' JavaScript. The SOP does not restrict JavaScript from interacting with different-origin sites in a number of ways which would be unlikely to interfere with their proper operation. For example, a script can modify its frame's document to fetch an image from any web site, which allows the script to communicate with the site through the name of the requested image. The SOP also allows scripts to use JavaScript's *intra-browser* channels to send messages to listening scripts from any origin. The result is that scripts that have access to confidential data can leak that data to cooperating outside web sites and JavaScript.

3. Challenges

BFlow requires a stronger policy than the SOP because it must prevent data movement even when untrusted scripts and untrusted servers collude against the user's wishes. BFlow must accomplish this while maintaining support for untrusted JavaScript extensions without encumbering deployment.

3.1 Threat Model and Security

BFlow applies to web sites that both store confidential user data and allow untrusted JavaScript to access that data. The adversary's goal is to read, with his own eyes, data that

[2] An origin is defined as a triple: domain name, protocol, and port.

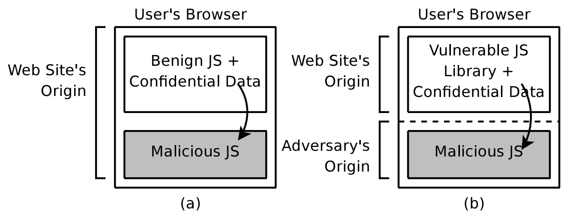

Figure 1: Malicious JavaScript reads confidential data (a) via the DOM and (b) by exploiting vulnerable JavaScript.

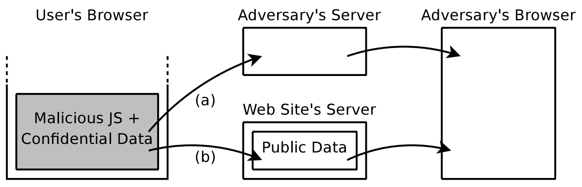

Figure 2: After reading confidential data, the malicious JavaScript leaks confidential data to an adversary via the (a) adversary's server (b) web site's public data.

he should not be able to read according to the web site's stated confidentiality policy. The adversary's capabilities are limited to creating his own accounts on the web site, running his own web servers, and writing JavaScript which the site includes in pages viewed by other users. Neither the site operators nor the users inspect the adversary's JavaScript.

More general adversaries might have other tools at their disposal. They might: compromise the host site; eavesdrop on or corrupt network traffic; infect the user's operating system with malware; infect the user's browser with malware; and use social-engineering attacks like "phishing" to lure the user or her friends into giving confidential data away. BFlow does not defend against these attacks, and its correct operation depends on adequate defenses to them that are outside the scope of this paper (e.g. SSL, timely application of O/S security patches, etc.).

The ability to inject arbitrary JavaScript into a page is quite powerful and is commonly referred to as a cross-site scripting (XSS) vulnerability. While BFlow does not aim to solve all attacks available through XSS, it does aim to prevent XSS attacks from leaking confidential data.

Attack Paths: Once the adversary injects JavaScript into the web site's pages and a user views a page, the JavaScript can attempt to read the confidential data displayed on the page and leak it to the adversary.

There are two possible scenarios for reading the confidential data. In the common case, the malicious JavaScript runs in the same origin as the confidential data. This could occur for many reasons; today, web sites incorporate large JavaScript libraries like Scriptaculous [script.aculo.us 2009] or Google Maps [Google 2009c] into their site's origin and platforms like Blogger inline completely unaudited third party scripts. In this case, the JavaScript can read the confidential data directly from the DOM as shown in Figure 1a. In the second case, malicious JavaScript can steal data even if there is no malicious code in the same origin as the confidential data. Today's browsers now support intra-browser communication between scripts from different origins and developers are already building libraries to use these channels [Ubl 2009]. If libraries like these are buggy, then malicious JavaScript running in the browser from a different origin (and a different frame) could exploit their bugs to read the confidential data as in Figure 1b.

After reading confidential data, the malicious JavaScript can send it to a web server using an HTTP request, either to the web site's own server or an external server. For example, the JavaScript can encode the data in an image name to be fetched from a server the adversary controls (see Figure 2a).

Even if the same-origin policy applied to all types of requests and the script could only send HTTP requests to the web site's server, the malicious JavaScript could leak data via the web site's own server. The malicious script could craft an HTTP request that stores the confidential data back onto the server in a public area. Since the server no longer realizes that the data is confidential, the adversary can read it with his own browser (see Figure 2b). Similarly, malicious JavaScript could write confidential data into a browser cookie and then any other code that comes from the same domain could read the data.

3.2 Flexibility and Adoption

The second challenge is to design a system that is easy for developers, web sites, and users to adopt.

One aspect of this challenge lies in preventing data leaks while preserving features popular among JavaScript developers, such as `eval()`, communication among concurrent browser scripts, and communication with remote web servers. This last JavaScript use is particularly commonplace and dangerous. Today's browser scripts routinely load images and data from multiple independently-administered servers. In the context of BFlow, such requests can encode confidential information. If one considers (as one should) a large majority of web servers to be untrustworthy receptacles for data leaks, BFlow must block requests (e.g, image loads) to such servers by scripts privy to confidential information. At the same time, BFlow can allow such requests from scripts that have not seen confidential data. In sum, BFlow should allow harmless requests to external servers, allow requests that release information if the release is the intention of the site owning the data, and detect and forbid accidental or malicious releases.

The design of BFlow should also be easy for users to install, site developers to adopt, and extension developers to adopt (in that order of priority). Some level of complexity is inevitable, but the goal is that deployment effort should be limited to: 1) users installing a browser plugin, 2) site developers deciding which data on their site is confidential

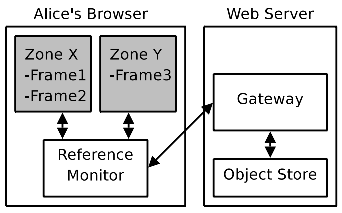

Figure 3: BFlow overview. Untrusted protection zones are shaded.

and rearranging the site's HTML to partition data by confidentiality constraints, and 3) third-party developers designing extensions that handle confidential data to live within BFlow's communication restrictions.

4. Design

The goal of BFlow is to enforce two properties on how a browser handles data. First, if confidential data arrives from a web site, only the human user and that origin web site should see any information derived from the data unless the site specifically allows it to go to another web site. Second, if the browser sends information derived from confidential data to the origin web site, the information must be marked as confidential unless the site specifically allows the removal of the confidentiality marking. The main tension in the BFlow design is the enforcement of these properties in a way that is compatible with how developers use JavaScript in complex web pages.

In outline, the BFlow design is as follows. The BFlow browser reference monitor watches how data flows into, out of, and within the browser. A BFlow-aware server sends a *label* along with data it sends to the browser to tell the reference monitor whether the data is confidential. The reference monitor uses a form of information flow control [Denning 1976] to enforce a confidentiality policy, tracking what data within the browser might be derived from confidential data. Each browser script runs in a browser frame, and frames are grouped into *protection zones*. BFlow tracks data at the granularity of a protection zone (see Figure 3). When data is about to leave the browser via the network, the reference monitor enforces a safety property on the data's label; if the data is going to its origin web site, the reference monitor includes the label; otherwise, if the label indicates the data is confidential, the reference monitor forbids its release unless an explicit *declassification* exception applies.

4.1 Information Flow Control

The BFlow reference monitor's information flow control system keeps track of what categories of confidential data the JavaScript in each protection zone may have seen. The reference monitor (RM) maintains a *label* for each zone. A label is a set of *tags*. A tag is an opaque token supplied by a server that indicates a particular category of confidential data. The meaning of a zone having a label containing a tag t is "the JavaScript or HTML in this zone may have observed information derived from data with confidentiality category t." A label with multiple tags indicates that the zone may have observed data in multiple confidentiality categories.

To ensure that a zone's label reflects the categories of confidential data it has seen, the RM enforces some rules relating to communication across zone boundaries. The effect of the rules is that, if information is to flow from zone S to zone R, R's label must be a superset of S's. In the special case of data flowing from a server to a zone, the zone's label must be a superset of the label provided with the data. Table 1 summarizes this and BFlow's other IFC rules described below.

A zone explicitly asks to change its own label and specifies which tags to add; BFlow does not automatically change zone R's label in response to the data R receives. BFlow always permits a zone to add any tag to its label. This is safe because the communication rules described above get strictly more restrictive as the sender's label grows. In practice, BFlow adds some further restrictions which we describe in Section 4.2. The RM imposes the IFC rules inside user u's browser to prevent buggy or malicious scripts from leaking u's data. At the same time, it is the server's responsibility to avoid sending data to u's browser that u is not permitted to read because u could have modified her browser to extract all the data available to it.

The ultimate source of each tag is a particular BFlow-aware web site. The browser RM internally adds the source server identity to each tag so that two tags from different servers are always unique. In typical use, a zone's label will either be empty (indicating that the zone has seen no confidential data) or contain just one tag. A label might contain multiple tags if a zone has consulted multiple categories of confidential data. A zone's label cannot contain tags from different web sites because it would violate the flow invariant described in Section 4.2

A web site decides what its tags mean. A typical web site might associate a different tag with each user, or a tag with each category of confidential data a user owns. For example, a web site might store both a confidential photo album and a confidential blog for user Alice, and associate a different tag with each kind of data. Then, if the site sends blog data to Alice's browser, and some JavaScript that examined the data communicates with the site, the site will know that the communication (and any resulting stored data) should have the same tag as Alice's confidential blog.

4.2 Protection Zones

One of the challenges in designing an information flow model for JavaScript comes from how developers use JavaScript today. Often, developers will construct web pages out of many sub-frames, each containing its own JavaScript. Furthermore, within a single page different sub-frames may have different purposes. For example, a top-level page may contain a chat tool and an email tool, each contained in its own individual sub-frame. Each of those tools may in turn

Sender	Receiver	Default Rule	Exception
Script in trusted zone	Any	Allow	N/A
Script in zone S, frame F, from server W	Script in W's trusted zone	Allow	N/A
	Script in zone S	Allow	N/A
	Script in zone R, sub-frame of F	$L_S \subseteq L_R$ (always true)	N/A
	Script in zone R, not sub-frame of F	$L_S \subseteq L_R$	Trusted zone proxy.
	Source server of W	Allow	N/A
	External server E	$L_S = \{\}$	$L_S \subseteq D_E$
Source server W sending data with label L	Script in W's trusted zone	Allow	N/A
	Script in zone R	$L \subseteq L_R$	None

Table 1: Default IFC communication rules and declassification exceptions; zones S and R are untrusted. The prototype implements these rules for communication through `postMessageBF`, the FID channel and HTTP requests, but it is more restrictive than these rules for shared DOM variables and cookie communication across zones.

contain its own sub-frames. For example, the chat tool may use two separate sub-frames, one for showing messages and one for data input.

Existing multi-frame modules like the chat tool typically read shared variables and call functions across frame boundaries. Modules expect these features to be reliable, so BFlow should accommodate this behavior; if one sub-frame in the module reads confidential data, then it should still be able to communicate with the other frames in the module without excessive coordination. BFlow addresses this challenge by applying IFC at the granularity of a protection zone.

A protection zone is a group of one or more browser frames, including their DOMs and the JavaScript running inside of them, plus its own set of browser cookies. All the scripts and data within a zone share a common label. Grouping frames into zones gives developers an easy way to modularize their scripts. Once the scripts are in a common zone, they can communicate with each other regardless of any label changes, even if a script in one of many sub-frames changes the zone's label unilaterally.

A web site also has a special *trusted* zone which always has an empty zone label; JavaScript running in the trusted zone can bypass BFlow's browser constraints. A web site uses the trusted zone in cases where confidential data is allowed to leave the system by a browser script, but such scripts must be carefully inspected.

To create a new zone, JavaScript in an existing zone requests a new zone id from BFlow and then loads a document from the server (specifying the new zone id) into one of the zone's existing frames. When the HTTP response arrives, the RM recognizes that the zone id is new, and creates its local representation of the zone. However, not all frames have their own zone; when a parent creates a sub-frame, by default the RM places the sub-frame in the same zone as the parent as shown by Z1 in Figure 4.

Flow Invariant: BFlow maintains a *flow invariant* over the browser's frames and zones: first, the browser's top level frame must be in the trusted zone and all its sub-frames must be able to legally send messages to the top level frame. Second, if a parent frame P has child frames C_i, then the

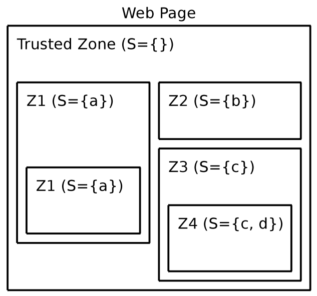

Figure 4: Web page frame hierarchy with zones and labels. Each box is a frame.

P must be able to send messages to each of its children legally. More specifically, if P has label L_P and P's children have labels L_{C_i}, then $\forall i, L_P \subseteq L_{C_i}$. This invariant must hold regardless of what zone each frame is a member of. The BFlow RM preserves the flow invariant by checking the target frame F and target zone Z before changing a zone's label.

When a zone Z changes its label, all other scripts running in Z will have the new label even if they are running in other frames; no zone other than Z will experience a label change. However, adding t to L_Z may permit another zone Z_P to add t to its label because of the invariant, if adding t to L_Z means all of Z_P's children now contain t.

Maintaining the invariant slightly limits the kinds of frame hierarchies possible: an untrusted frame cannot contain tags from different web sites and a parent frame with $L_P = \{t\}$ cannot contain a child frame with $L_C = \{\}$, but it ensures that BFlow can support existing methods of JavaScript communication described in Section 4.3.

4.3 Controlling Intra-browser Communication

Tracking the flow of confidential data between scripts within the browser is critical to preventing leaks because BFlow can only prevent a script from leaking data if it knows what data the script has seen. This section describes which channels are available in BFlow between scripts in the same zone and in different zones. We focus on the Firefox 3.0

browser in which JavaScript has four techniques to communicate between scripts (other browsers may have other techniques). They are DOM variables, browser cookies, the `postMessage` channel, and the fragment-ID (FID) channel.

Within One Zone: BFlow need not restrict communication between two scripts in the same zone, since all of the JavaScript, frame DOMs, and cookies within a zone share the same zone label. It is important that BFlow accommodates scripts from different frames that read and write each other's DOM variables, since many sites have scripts that use that feature.

Between Two Zones: Since two scripts in different zones can have different labels, BFlow must restrict communication between two such scripts according to the IFC rules shown in Table 1. It does so through a combination of unconditionally forbidding some operations between scripts from different zones, and allowing other operations only when the zone labels allow.

Although today's browsers allow scripts in the same origin to read and write each other's DOMs, BFlow unconditionally forbids JavaScript in two different zones from reading or writing each other's DOM variables or cookies. This is a conservative restriction due to our implementation and the only restriction BFlow places on code that has not seen confidential data. A better implementation would allow a script in zone S to write to variables and cookies in zone R if R's label were a superset of S's label.

Instead of using shared DOM variables and cookies, BFlow allows scripts in different zones to send explicit messages to one another using an API function called `postMessageBF`. To preserve the IFC rules, the RM only delivers the message if the sender's label L_S is a subset of the receiver's label L_R; if not, it will drop the message. BFlow's `postMessageBF` replaces the `postMessage` API found in HTML5 because `postMessage` does not enforce the IFC rules.

The fourth intra-zone communication method is the FID channel which is an artifact of a script's ability to set the location of both its sub-frames and the top level frame. Setting the location of frame F communicates data to frame F [Barth 2008]. BFlow does not specifically restrict the FID channel; instead, BFlow ensures that any use of the FID channel is legal according to the IFC rules in Table 1 because BFlow preserves the flow invariant. Without the invariant, a sub-frame P with label $L_P = \{t\}$ that has read confidential data could leak it to a child frame C with $L_C = \{\}$ that does not have the proper label, i.e. $L_P \not\subseteq L_C$.

These IFC rules alone might be too strict for an untrusted script that handles both confidential and public data, and also needs a way to reveal the public data. For example, an untrusted script might need to read a user's confidential email address with label $L = \{t\}$ and also need to save public data with $L = \{\}$ to the server. BFlow supports this using an exception to the strict IFC rules called *browser declassification*. BFlow permits a script running in a zone from server W to send messages to scripts in the trusted zone of server W and vice-versa, so the untrusted script with label $L_R = \{\}$ can request the email address from the trusted script and the trusted script can respond with the email address despite R's label if the site developers allow it to.

4.4 Controlling Browser-Server Communication

In addition to data flowing within the browser, data can also flow between the browser and web servers in HTTP requests and responses. To track these flows, the BFlow reference monitor interposes on requests sent out by the browser and on responses that arrive at the browser. When handling an HTTP request from a zone that has seen confidential data from server W, BFlow treats the *source* server W differently from any other *external* server E_i. Since W sent the confidential data in the first place, BFlow can safely send HTTP requests containing the confidential data back to W. Sending to any other server E_i requires a declassification exception, whether E_i is BFlow aware or not.

Source Server Protocol: For communication between the browser and the source server, the BFlow RM and the server include labels in each HTTP request and response. The server labels responses so that the browser RM will know what label to apply to each zone. Similarly, the browser RM labels requests so that the server will know what data is confidential otherwise, attacks like that shown in Figure 2b might succeed.

When a browser script makes an HTTP request, the BFlow RM sets the label of the request equal to the script's zone label, i.e. $L_{req} = L_{zone}$. Labeling the request according to the script's label ensures that the server will know what confidential data the request may contain. If the request causes the server to store data, the server should store the label along with the data and return the label if a subsequent request reads it.

By default, the server's HTTP response will have the same label as the request ($L_{req} = L_{resp}$). This ensures that any confidential data contained in the request will propagate to the response and the label of the zone that receives the response will reflect the confidential data in its label. To avoid inappropriately leaking confidential data, the server should not use any data with tag t to generate the response unless the response's label will contain t.

Also, since any user's browser can ask to add t to a zone's label (including users who do not have permission to read data with tag t), before sending data with tag t to the browser, the server first checks whether the user logged into the browser has permission to read the data.

In addition to asking the RM directly, a script can also add a tag t to the target zone's label as part of an HTTP request. This allows a parent frame to load a page into one of its sub-frames in a different zone with a different label. It is shorthand for first loading a script into the sub-frame, having the sub-frame change its own label and then requesting the additional confidential data. The server then adds t to the response's label $L_{resp} = L_{req} \cup \{t\}$. This method only works if the frame that makes the request has permission to load a page into the target frame which implies that the requester can send a message to the target; either the two frames are in the same zone, or the target frame is a sub-frame of the requester.

Propagating the information flow labels to the server and back ensures that the client cannot leak data by bouncing it off the server. In IFC terms, if a script in zone X tries to send data to zone Y via an HTTP request through the server, the RM will update Y's label with the server response's label $L_Y \leftarrow L_Y \cup L_{resp}$ and therefore the communication will abide by the IFC rule $L_X \subseteq L_Y$.

External Servers: BFlow forbids communication from scripts that have seen confidential data to external servers, conservatively assuming that they are not trustworthy. This applies both to image loads and to AJAX requests. The RM permits a script to send a request to an external server if the script has not seen confidential data.

This rule is too restrictive for some web sites. Applications such as mashups may need to request data from external servers in a way that the request itself necessarily leaks confidential information. In such sites, the developers can create a *request declassification* rule which allows certain kinds of confidential data to exit to certain external servers.

For example, a web site W might want to fetch the weather forecast for a user based on the user's postal code even though the postal code is confidential. If W's developers trust the weather server E enough to reveal its users' postal codes, then W can add a request declassification rule that says "any data tagged with tag t_i may be sent to E" and BFlow will permit scripts that have read data with t_i (but only t_i) to send HTTP requests to E. More precisely, the site administrator would add t_i to E's declassification set D_E (see Table 1).

5. Visible Model

Developers and users must understand some aspects of BFlow.

5.1 Developer Visible Model

Labels: An application developer must create a labeling scheme for the application's data, an arrangement of the application's HTML and scripts into frames and zones, and plan for labeling the zones. Zone labels are usually predictable: for example, the developer knows that a certain frame will display the user's confidential postal address and that its zone will always have exactly the corresponding label. This predictability prevents unexpected increases in labels and surprise violations of BFlow's rules.

How many tags a site uses and what the tags correspond to are largely application-specific, and BFlow does not prescribe any particular approach. In general, for each collection of data that some users and/or some external sites should be able to see, but others should not, it is likely that a tag should be associated with that data. Many sites will have a handful of tags for each user, for example one for the user's contact details and one for the user's confidential blog.

Frames: A typical BFlow web page will consist of several frames. The top level frame will always be in the trusted zone. It will have sub-frames, each with a zone and label, to contain untrusted scripts. Scripts that need to see different kinds of confidential data will be in separate zones. A particularly common case will be separate frames that display images from external servers but handle no confidential data, and frames that handle confidential data. Existing applications may need to re-factor their HTML in order that scripts that handle data with different confidentiality tags are in separate frames and zones.

As an example, a page that allows a user to edit both his confidential phone number and his public personal profile would contain two frames in separate zones: one containing the phone number, and one containing the personal profile. Because the zones are separate, the user can edit his profile without the risk of a script reading the confidential phone number and inserting it into his public profile.

Data that the user enters into a form field takes on the label of the zone surrounding the field. Thus, even if a frame does not initially contain confidential data, if the frame contains a form field into which the developer knows the user may enter confidential data, the developer should put the field in an appropriately labeled zone.

Developers can also privilege-separate large pieces of code into a small portion running in a trusted zone and a large portion running in an untrusted zone. The two portions can communicate using browser declassification. For example, the trusted portion could provide a limited API to access external web servers.

Linking: If an untrusted page has not seen confidential data, it can link to external web sites, but if it has seen confidential data, it can only link to external web sites if the destination server has a request declassification rule.

Since the top level frame in a BFlow web page must be in the trusted zone, when an untrusted page with label $L = \{t\}$ loads a new page into the browser's top level frame, the BFlow does not propagate tag t to the top level frame. Since this is equivalent to declassifying the t tag, the trusted page should not transmit any unique data from the HTTP request such as `POST` parameters to an untrusted frame unless its label also contains t.

Confidential Data and External Servers: As described in Section 3.2, today's browser scripts sometimes load images and data from external servers after seeing confidential data.

One example of this is a confidential blog page that loads a static background image from an untrusted photo web site

E. Since the HTML contains confidential data and JavaScript, BFlow cannot determine if the request for the image has been influenced by confidential data or not. If the script requested the image after computing on the confidential blog content, the HTTP request would be leaking data to E. However, in this scenario, the image that the page is loading is static and is not based on the confidential data. To build such a page, the site developer can pre-declare a set of external web documents which BFlow prefetches directly from the external servers and then caches on the blog's server. Since the requests have not been influenced by confidential data, they will not leak any data to the external servers. When the browser loads the image, it fetches it from the blog server, not the photo server E, thus decoupling the request made by the browser from the request that arrives at the photo server and protecting the blog's confidential content.

Prefetching does not work for all web applications: a script may not know what data it needs until after reading confidential data, or the potentially-needed data may be too large to prefetch. For example, a mashup script that displays a user's location on an externally-fetched map will not know what map images to fetch until after it reads the confidential address. In this type of mashup, BFlow cannot protect the privacy of the addresses from the map server. However, keeping the address confidential is an unrealistic security requirement because the map server cannot function efficiently without the address. A more realistic security requirement is that the mashup only sends the confidential address to the map server, and not to other external servers. BFlow can enforce this requirement using request declassification as described in Section 4.4.

Script Changes: Depending on the web site, untrusted scripts and libraries may or may not need to understand the information flow system. For some web sites, the site programmers may be able to determine what label an untrusted script should run with, so that the untrusted script need not be aware of BFlow. For example, if a web site imports a JavaScript library like Scriptaculous [script.aculo.us 2009] and never expects the library to contact external servers or communicate with different zones, the site could just use the correct non-empty label and import the library without modifications. For scripts that only read data and render it to the user, the site can just load the script with a label containing all the tags the user can read.

Server Code: A server that supports BFlow scripts must be able to record the label of data arriving from a script, and emit that label when it later serves the same data to a script. A straightforward approach is to store a label with each file or database entry. Though not necessary, it might also be helpful for the server to use an IFC-aware operating system or server framework [Efstathopoulos 2005, Zeldovich 2006, Krohn 2007a].

Debugging: To debug applications written for BFlow, developers test their HTML and JavaScript in a BFlow-enabled browser which reports error messages pertaining to BFlow's information tracking system.

5.2 Users Visible Model

End users interact with a BFlow site much like they do with web sites today. Depending on the web site, a user may need to understand that a sub-frame may have a different privacy policy from the rest of the page. For example, a web site that includes confidential content may also include an untrusted JavaScript widget running in a sub-frame that has not read confidential data. In this case, it is the web site's responsibility to indicate to the user that any data he types into the sub-frame may be visible to the public. This responsibility is more explicit in BFlow, but it already exists in any web site that includes content from untrusted programmers whether using sub-frame isolation or not.

6. Implementation

BFlow requires browsers to confine browser JavaScript into protection zones and to exchange security metadata with servers in each HTTP request. Since today's browsers do not implement these features, and replacing the installed base of web browsers is difficult, the major challenge in implementing BFlow is making it easy to deploy to browsers.

6.1 Client Implementation

To ensure that our BFlow client modifications are easy to install for end users, we implemented the client-side reference monitor as a Firefox 3 plugin. The plugin is a portable JavaScript and XML package that runs on any platform that supports Firefox 3; users can install the plugin with only two mouse clicks. Firefox does not provide many security related hooks in the plugin interface, but it does implement the same-origin policy which provides fairly strong isolation between different origins.

The BFlow plugin takes advantage of the existing SOP in the browser to implement basic isolation between protection zones. It associates each zone with a unique unforgeable domain name, and each different BFlow web site has its own disjoint set of zone domain names. Zone domains are of the form `Z.site` where `Z` and `site` are the respective unique names of the zone and web site. BFlow uses the form `Z.site` rather than `Z.site.com` because browsers permit a script to remove its host prefix from its domain name before the SOP comparison; using `Z.site.com` would allow two scripts with zones `Z1.site.com` and `Z2.site.com` to remove `Z1` and `Z2`, and thus communicate based on the common name `site.com`.[3] Separating zones into different domains uses the SOP to prevent scripts in one zone from reading and writing DOM variables and cookies in another zone.

[3] The RM uses Firefox's SOP implementation, so it handles domains like `cnn.co.uk`.

However, the SOP alone does not prevent JavaScript in two different zones from colluding to leak confidential data; a script in one zone can communicate with a script in another zone using cross-domain channels like the fragment-ID channel and `postMessage` described in Section 4.3. BFlow's Firefox plugin disables `postMessage`, and the flow invariant described in Section 4.2 ensures that all available fragment-ID channels in Firefox 3 are also legal data flow paths according to BFlow's information flow rules. The BFlow prototype relies on the FID descendent policy in Firefox 3 and other recent browsers that limits the channel to parents sending data to children and frames sending data to the top-level frame [Barth 2008].

When the browser makes an HTTP request to a zone domain on a BFlow aware server W, the browser RM directs the request to a web proxy server running on W which then forwards it to an Apache web server process on W. Using a proxy prevents the browser from attempting to resolve the zone's DNS name which is not an actual DNS domain name; however, the proxy is specific to our prototype and the same functionality could be built into the web server.

The browser plugin is 1003 lines of JavaScript and 89 lines of XML including comments. To intercept HTTP requests for inspection and modification we use Firefox's "http-on-modify-request" and "http-on-examine-response" hooks in its XPCOM observer service. These hooks are called before sending each HTTP request and before returning the response to the rendering engine respectively.

6.2 User Authentication

A user can initially authenticate himself to a BFlow site using any technique, but any script used in a login web page should be a trusted script. It could be possible to use an untrusted script on the login page with a tag to protect the password data, but the site would need to generate a new tag for each login attempt, or else a script could transmit the username and password to another user that attempts to log into the system later.

After logging in, the user authenticates each subsequent HTTP request using an authentication cookie. The cookie is confidential data, but BFlow does not protect it using the information flow system because the browser must authenticate the user for all HTTP requests, even requests for public data where $L = \{\}$, so the cookie cannot have its own tag, otherwise a public page would also be protected by the cookie's tag. Instead, BFlow associates the cookie with the web site's real domain name, for example, `site.com`.

Untrusted JavaScript running in a protection zone cannot read the web site's authentication cookie because the untrusted zone's domain is of the form `Z.site` and the authentication cookie is from the domain `site.com`. Since the domains do not match, or share a suffix, the same origin policy prevents the untrusted JavaScript from reading the authentication cookie. However, a standard browser will not send the authentication cookie for requests originating from `Z.site` for requests to `site.com` because of the SOP, so the BFlow RM attaches the cookie to these HTTP requests.

6.3 Server Implementation

In the BFlow prototype, the server implements the interface described in Section 4.4 with server processes called gateways. The client sends raw tag values to the server in the headers of each HTTP request, and the server response with tag values in the response headers.

The server uses a gateway process to handle each request which in turn invokes application logic. The gateway launches the application logic with the read privileges of the user, so it can only read the data that the end user may read. This ensures that the user will not receive data he does not have permission to read.

Although it is not necessary for a BFlow server to use an IFC operating system, the prototype's gateways and application logic both run in the Flume IFC system [Krohn 2007a] which provides IFC within the Linux operating system. Running the application logic in an IFC OS has the advantage that untrusted code can safely run both in the client and in the server in a unified IFC space.

At a lower level, each gateway is a long-running Python FastCGI process. The gateway serves static files directly off the file system and queries application request handlers, which are Flume-confined FastCGI processes, to serve dynamic HTTP requests. The gateway is 4144 lines of Python including comments.

6.4 Server Storage

As described in Section 4.4, a BFlow server can allow untrusted scripts to store data on the server as long as the server associates a label with the data when writing and reading. The BFlow server prototype implements a key-value storage system within its IFC environment. Untrusted browser scripts can read and write data to server storage using AJAX HTTP requests.

When an AJAX request stores data on the server, the storage system labels the data with the label of the request. Later, when an HTTP request reads that data, the storage system only reads data whose label is a subset of the HTTP response's label. The underlying storage system is an IFC database wrapper built on top of PostgreSQL that resembles the SeaView [Lunt 1990] data model.

Although the prototype storage system runs in an IFC operating system, it is not necessary to use one. In many cases, it should be sufficient for the server to store a label alongside the data and apply the label when reading the data. Together the IFC database wrapper and the HTTP storage request handler are 3288 lines of Python including comments.

7. Applications

To demonstrate that BFlow preserves privacy and is flexible enough to build web platforms, we implemented two web applications within the BFlow framework and a collection of untrusted JavaScript extensions.

7.1 BF-Blogger

Blogger [Blogger 2009] is a popular blog hosting service that supports confidential blogs that only specific users can read. Blogger allows a blog's author to install third-party JavaScript extensions that run in the browsers of all viewers of the blog. These extensions can use confidential data, such as recent posts in the current blog. Other extensions talk to external web servers: for example, one extension displays random images from a photo-sharing web site. All JavaScript runs in the same browser frame with access to the blog's confidential data, including the blog posts and the reader's browser cookies making it possible for malicious scripts to leak the data.

BF-Blogger is derived from Blogger's HTML, JavaScript, and third party extensions, but it runs in BFlow. In a BF-Blogger blog, the top-level trusted zone contains one child and protection zone for the main blog content (including Blogger's JavaScript) and a separate child and zone for each extension. BF-Blogger associates the data from a confidential blog with tag t.

The main blog content's zone contains the blog's confidential content, so it starts with the label $L = \{t\}$. Each extension zone starts with an empty label $L = \{\}$. An extension can make an HTTP request to the server to read confidential blog contents, thus changing its label to $L = \{t\}$.

We ported seven Blogger extensions to BF-Blogger. The *Twitter* and *Flickr* extensions fetch data from external web servers; they do not read the confidential blog contents, so BFlow permits them to fetch the external data. The *Recent Posts* extension fetches the current blog's contents, computes a set of post snippets, and displays them to the user. The *Cbox* extension implements a multi-user chat room. Cbox consists of multiple cooperating frames, each with its own JavaScript and the individual frames read and write the other frame's DOM. BF-Blogger runs Cbox as if it had read confidential data ($L = \{t\}$) because it stores data on the server, and users might chat about the confidential blog contents. Cbox consists of multiple frames, but since BF-Blogger groups them into a single protection zone, BF-Blogger can set the zone label just once. This changes the label for all of Cbox's frames without BF-Blogger being aware of all of Cbox's sub-frames. Because the chat contents might be confidential, we modified Cbox to store its data in BFlow server storage with label $L = \{t\}$. We also wrote two *Evil* extensions that run in both Blogger and BF-Blogger; their goal is to leak data from a confidential blog (see Section 8.1.2).

Extension developers for BF-Blogger need not understand the details of BFlow other than that they may not make external HTTP requests after reading confidential data.

7.2 BF-Socialnet

BF-Socialnet is a multi-user social network that uses BFlow to protect privacy. Each user has a profile and a set of friends. BF-Socialnet permits JavaScript extensions to run within its pages with access to the user's profile and friend list. We implemented two JavaScript extensions, a *profile comparison* tool and a *messaging* tool to exercise BFlow's support for different communication patterns and privacy policies.

BF-Socialnet's base *friend* privacy policy is that user Alice's profile and friend list is only visible to Alice's friends. In addition, BF-Socialnet supports *personal* data which only Alice may read and *pairwise* data that a particular pair of users may read. To implement these policies, BF-Socialnet uses a set of tags for each user, one tag for personal data that only Alice can see (t_{alice}), one tag for the Alice's friend-visible data ($t_{alice:friends}$), and one tag for each of Alice's friends for pairwise-visible data; for example if Alice is friends with Bob, BF-Socialnet would use the tag $t_{alice:bob}$.

The BF-Socialnet page has a trusted root page that contains different sub-frames for each third party extension. The root page has multiple frames for each extension, each with a different confidentiality mode. For example, in one frame, the messaging extension runs in a mode that allows it to read all data that the user can read. In a separate frame, the messaging extension runs with a pairwise tag determined by the root page. The user selects who to send a message to using a drop down box in the root frame, and the root frame adjusts the label on the frame accordingly. The profile comparison tool only reads data, and therefore only runs in a mode that allows it to read all data that the user can read. It uses AJAX requests to read the profiles of all the user's friends, compares them in the browser, and outputs a list of friends with similar interests.

User and Developer Visible Model: In BF-Socialnet, an application writer needs to know what confidentiality mode his application will run under and what data it hopes to read. However, he does not need to understand labels, tags, or the information flow model. Similarly, users should be able to understand that the different sub-frames abide by different confidentiality modes because data that they input to a sub-frame will abide by the frames confidentiality mode. This decision is similar to the decision that users make currently when choosing their profile's privacy policy, so we expect users will be able to understand it.

8. Evaluation

This section evaluates how well BFlow achieves its two main goals: prevention of confidential data leaks from in-browser JavaScript, and compatibility with existing developer uses of JavaScript. We focus on these topics rather than performance because the performance penalty of the browser extension should be minimal and the HTTP proxy can be eliminated by moving its functionality into the web server.

8.1 Security

8.1.1 Attack Analysis

This section explains how BFlow prevents the example attacks described in Section 3.1, Figures 1 and 2.

In Figure 1a, malicious JavaScript resides in the same frame (and thus the same zone) as the confidential data. BFlow ensures the a zone's label includes tag t before it allows the zone to read confidential data with tag t, therefore the malicious script will be running in a zone with tag t. This label constrains the malicious script so that it can display data only to the browser's human reader and the source web server. The former is not a leak, since the source server would not have sent the data unless the browser's user had permission to read it. The latter is not a leak because BFlow propagates tag t along with the data, so that the source server will know it is confidential.

In Figure 1b, the confidential data (and benign JavaScript) is not in the same zone as the malicious JavaScript. If the benign JavaScript accidentally tries to communicate with the malicious JavaScript, the BFlow reference monitor will forbid the communication unless the malicious JavaScript's zone's label is a superset of the label of the zone with the confidential data. In the latter case the malicious JavaScript will be restricted from leaking as described in the previous example.

8.1.2 Attack Examples in Blogger

In order to verify that BFlow fixes existing security problems, we implemented two JavaScript extensions for Blogger that steal confidential information.

The first extension contains a cross-site scripting (XSS) attack that exploits a typical script injection vulnerability. We wrote this attack, but we believe that XSS attacks in the wild would use the same leak technique since today's web sites do not usually use any counter measures. In this attack, the adversary tricks user A into placing the extension on his blog so that viewers of his blog execute the extension's script. When some user B views A's blog, the extension reads user A's confidential blog contents and user B's Blogger cookie and sends it to an external server using an image request, thus leaking A and B's confidential data. This attack works when run on the real Blogger web site, but the extension is unable to leak data when run on BF-Blogger, since BFlow forbids the extension from contacting the external server because its zone has seen confidential data.

The second attack is meant to approximate the one pictured in Figure 1b. We believe this is a new style of attack and are unaware of such attacks in the wild because intra-browser JavaScript APIs are currently uncommon. The attack consists of two parts: the *listener* and the *leaker*. The leaker takes the place of a vulnerable script API and the listener takes the place of an adversary that tricks the vulnerable script into reading confidential data and sending it to the listener. In this attack the listener script resides in a frame in the adversary's origin, and listens for a message from the leaker. The leaker runs in the same origin as the confidential Blogger page, and sends confidential data to the listener using `postMessage`. Again, this attack works when run on the real Blogger web site, but the leaker is unable to send data to the listener with `postMessageBF` in BF-Blogger, because BFlow forbids the leaker (who has seen confidential data) from messaging the listener (who has an empty label $L = \{\}$).

Extension	LOC	LOC Included	LOC Changed	Confidential Data?
Twitter	6	19	0	No
Flickr	10	0	0	No
Buzz	1	0	0	No
Blogger JS	60	851	0	No
Youtube	1282	610	0	No
Calendar	804	1141	0	No
Weather	2993	797	0	No
Popular Posts	16	0	1	Yes
Commenters	15	0	1	Yes
Recent Posts	9	65	2	Yes
Random Post	34	0	2	Yes
CBox	801	0	89	Yes

Table 2: Lines of code (LOC) changed to port existing widgets to BF-Blogger and whether they see confidential data.

8.2 Adoption

In order to evaluate the complexity of developer adoption, we ported several existing Blogger widgets [Beautifulbeta 2009, Twitter 2009, Flickr 2009] to BF-Blogger. They fall into three categories:

- Those that load data, images, or libraries from external servers, or link to external servers.
- Those that read the blog's confidential content using the blog's JSON feed.
- Those that do both of the above.

Extensions in the first category, such as the *Flickr*, *Twitter*, and *Buzz* extensions required no changes to work on BF-Blogger. These extensions need no confidential data, so they can be loaded in frames that have an empty label, and are free to fetch data from external servers.

The *Recent Posts* extension is in the second category. It fetches the blog's most recent posts and displays a list of them on the blog's side bar. The original version loads a JavaScript file from an external site, which fails because the script reads the blog content before making the external HTTP request for the JavaScript file. To make this extension work in BF-Blogger, we copied the content of the external JavaScript file into the extension.

The two extensions we found in the third category, namely *Popular Posts* and *Top Commenters* are a form of mashup. They use an external server (Yahoo Pipes [Yahoo 2009]) to process the content of the blog's confidential comments and then display the results in the page. They illustrate how a mashup sometimes trusts an external server with confidential data. To add support for these in BF-Blogger we added a comment feed to the blog and made the feed available to only the Yahoo Pipes client host. This feed policy is an explicit declassification of the confidential comments to the Yahoo Pipes host.

We also examined a number of Google Gadgets [Google 2009a]. The twenty most popular Google Gadgets don't act on confidential data, and just import data from external sites or from Google's platform. We ported the generated JavaScript of three Google Gadgets to run on our platform: *Youtube Search*, *Google Calendar*, and *Current Weather*. All worked without changes.

The *Cbox* messaging system required more code changes since it stores persistent data to the server; it was modified to read included files from our platform and to store messages using our server storage API.

9. Limitations

BFlow has a number of limitations; it does not support all kinds of web page designs and it does not protect against all types of attacks.

One page design limitation we are aware of is due to the coarse grained nature of BFlow. Since BFlow only tracks data at the granularity of frames, a single untrusted browser frame cannot simultaneously handle confidential data and public data without marking the public data as confidential. In order to protect the confidential data, a BFlow application would label the frame with $L = \{t\}$, but then the public data would also be labelled with $L = \{t\}$ and be unavailable to the public. This is a scenario where finer grained information tracking [Myers 1997] would help. Site developers might also have to refactor their HTML to partition data into frames to separate confidential data with different tags.

Users might also be confused that frames have different security labels and type sensitive data into frames with $L = \{\}$ which would leak the data. Web sites can help by marking frames, but BFlow does not provide a solution for this.

When designing a label based confidentiality scheme, reasoning about labels is not always straightforward and errors in designing a scheme can result in data leaks. BFlow does not provide assistance for using labels, but other projects hold promise [Efstathopoulos 2008].

Another limitation of BFlow is that it does not apply to browser plugins. For example, BFlow does not support Flash [Adobe 2009] or Java [Gosling 2005] plugins.

There are a number of attacks for which BFlow does not offer a solution, including covert channels [Lampson 1973] and phishing. If a malicious script with label $L = \{t\}$ uses a covert channel like CPU modulation to send data to a script with label $L = \{\}$, it can leak the confidential data. If a malicious script uses a phishing attack to trick a user into revealing his password the attacker can subsequently login as the user and read all his confidential data.

As described in Section 3.1, BFlow does not protect against a compromise in the servers, browsers, operating systems, or the BFlow software itself. For example, if an attacker can trick a user into installing his malicious Firefox extension, he could disable BFlow. Similarly, web sites with weak user authentication are vulnerable in ways that BFlow does not fix.

If an attacker is able to cause a trusted zone in BFlow to load and run his malicious code, then the script will act with the privileges of the trusted zone and will be permitted to leak confidential data. However, trusted zones are intended to be very carefully validated and to never run third-party code; BFlow protects data in all non-trusted zones from leaks.

10. Related Work

One way to understand existing work is in two broad categories: discretionary access control (DAC) (including capabilities-based systems and least-privilege isolation techniques) and mandatory access control (MAC) (including language-based and runtime IFC).

Works like Tahoma [Reis 2007], Google Chrome [Google 2009b] and MashupOS [Wang 2007] and Caja [Miller 2008] all fit the DAC model. Tahoma isolates applications from each other using virtual machines so that even buggy browsers running malicious code cannot tamper with cookies or DOM objects in other browsers. Users can choose to share data across web sites with explicit whitelists of all other hosts that can be contacted as the page is rendered and as the JavaScript (or other plugins) run. Thus, Tahoma offers all-or-nothing sharing at the discretion of the original web site; it does not allow a web site to safely give confidential data to potentially malicious scripts. The Chrome browser implements the same style of isolation between browser windows, but with process-based rather than VM based isolation.

MashupOS proposes changes to web browsers and servers to isolate third party JavaScript code with more flexibility than today's browser frames and finer granularity than inlining scripts today. MashupOS proposes HTML extensions such as `<Sandbox>` and `<OpenSandbox>`, which occupy a middle ground: they allow the caller and callee to communicate but only along well-understood channels (as opposed to across the whole DOM under the status quo). However, MashupOS has the same limitations that DAC-based operating systems have: the user (or the *integrator* in MashupOS's terminology) must still decide *a priori* whether to trust a third party or not with sensitive data because sandboxed scripts in MashupOS can leak data to external servers. In BFlow, untrusted scripts can decide whether to read private data at runtime.

Other works like Caja follow MashupOS's lead. Caja confines a subset of JavaScript into an object-capability model. As in MashupOS, the goal is to allow finer-grained sharing of data between cooperating browser components.

By contrast, MAC systems allow untrusted software to compute with confidential data, while preventing that software from exposing it. MAC has long been a technique at play in programming languages [Denning 1976] and operating systems [Bell 1976, McIlroy 1992, Dep 1985], which modern research [Efstathopoulos 2005, Zeldovich 2006, Krohn 2007a, Myers 1997] suggests is practical for server-side web applications. The same tools apply in the context of browser-based security.

The SIF system [Chong 2007b] uses language-based information flow control to maintain privacy constraints between browser and server, but assumes no malicious or buggy JavaScript. The Swift system [Chong 2007a] uses IFC to automatically split web applications into trusted server-side Java and untrusted browser-side JavaScript. BFlow applies similar information control analysis, but at runtime. BFlow retains a similar correctness property, that code will produce a fail-stop error instead of leaking data. While Swift only applies to JavaScript output by the Swift compiler, BFlow's reference monitor applies to all JavaScript code, such as legacy and hand-written libraries. However, BFlow does make trade-offs; firstly, it has coarser-grained security compartments (browser zones) while Swift tracks information flow per variable. Secondly, BFlow requires users to install a browser plugin and Swift-like system would not. Using a browser plugin enables BFlow to ease the adoption burden placed on site developers at the expense of the end users.

Vogt et al. [Vogt 2007] also track information flow control at runtime to prevent cross-site scripting attacks. However, they have limited their system to client-side changes only, and therefore cannot prevent attacks that move data back and forth between the browser and server. Spectator [Livshits 2008] tracks taint between browsers and servers, but its goal is to detect JavaScript worms, not protect privacy.

Other work proposes curtailing JavaScript's power to solve traditional XSS problems. BrowserShield [Reis 2006] rewrites arbitrary (potentially malicious) JavaScript to a safer core. BEEP [Jim 2007] firewalls unsafe JavaScript by limiting which servers it can contact as it executes. Hallaraker et al. [Hallaraker 2005] audit JavaScript execution, and use intrusion-detection techniques to sense anomalous execution patterns. These veins of work show promise against traditional XSS attacks but do not handle data leaks which involve sending data back and forth to the origin server.

A complementary way to build web extensions is on the server-side, rather than on the browser. Facebook [Facebook 2009] and OpenSocial [Google 2009d] give third-party developers access to server-based data, allowing them to customize and extend existing server-based features. The Menagerie [Geambasu 2008] system presents an interface to make server data more accessible. All of these systems use discretionary security controls, requiring users to either trust or reject third-party code. W5 [Krohn 2007b] proposes to achieve similar features with MAC, but a W5 implementation would need to solve the security challenges discussed in Section 3 to allow third-party server-side extensions to push unvetted JavaScript to browsers.

11. Conclusion

Many of today's web sites currently use JavaScript that they might not understand, including large libraries and third-party extensions. The combination of these possibly buggy or malicious scripts and confidential data leaves that data open to attack. BFlow is a novel browser based information flow control system that allows mostly unmodified legacy JavaScript to read, compute with, and write confidential data without the risk of compromising user privacy.

Acknowledgments

The authors thank Micah Brodsky, Frans Kaashoek, Chris Lesniewski-Laas, Michael Walfish, Nickolai Zeldovich, the anonymous reviewers, and shepherd Fred Schneider for their comments on drafts of this paper. This work was supported by a National Science Foundation fellowship and Nokia.

References

[Adobe 2009] Adobe. Flash. `http://www.adobe.com/products/flash`, Jan 2009.

[Barth 2008] Adam Barth, Collin Jackson, and John C. Mitchell. Securing browser frame communication. In *Proceedings of the 17th USENIX Security Symposium*, pages 17–30, San Jose, CA, USA, July 2008.

[Beautifulbeta 2009] Beautifulbeta. Blogger widgets. `http://beautifulbeta.blogspot.com`, Jan 2009.

[Bell 1976] David E. Bell and Leonard La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, USA, Mar 1976.

[Blogger 2009] Blogger. Site. `http://www.blogger.com`, Jan 2009.

[Chong 2007a] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 31–44, Stevenson, WA, USA, Oct 2007.

[Chong 2007b] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium*, pages 1–16, Boston, MA, USA, Aug 2007.

[Denning 1976] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[Dep 1985] *Trusted Computer System Evaluation Criteria (Orange Book)*. Department of Defense, dod 5200.28-std edition, Dec 1985.

[Efstathopoulos 2008] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 301–313, Glasgow, Scotland, 2008.

[Efstathopoulos 2005] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, Brighton, UK, Oct 2005.

[Facebook 2009] Facebook. Site. `http://www.facebook.com`, Jan 2009.

[Firefox 2009] Firefox. Add-ons. `https://addons.mozilla.org/`, Jan 2009.

[Flickr 2009] Flickr. Badge. `http://www.flickr.com/badge.gne`, Jan 2009.

[Geambasu 2008] Roxana Geambasu, Cherie Cheung, Alexander Moshchuk, Steven D. Gribble, and Henry M. Levy. Organizing and sharing distributed personal web service data with menagerie. In *Proceedings of the 17th International World Wide Web Conference*, pages 755–764, Beijing, China, Apr 2008.

[Google 2009a] Google. Gadgets. `http://www.google.com/webmasters/gadgets/`, Jan 2009.

[Google 2009b] Google. Google chrome: a new web browser for windows. `http://www.google.com/chrome`, Jan 2009.

[Google 2009c] Google. Maps API. `http://code.google.com/apis/maps`, Jan 2009.

[Google 2009d] Google. Open Social. `http://code.google.com/apis/opensocial`, Jan 2009.

[Gosling 2005] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[Hallaraker 2005] Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in Mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, Shanghai, China, Jun 2005.

[Jim 2007] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 601–610, Banff, Alberta, Canada, May 2007.

[Krohn 2007a] Maxwell Krohn, Alex Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 321–334, Stevenson, WA, USA, Oct 2007.

[Krohn 2007b] Maxwell Krohn, Alexander Yip, Micah Brodsky, Robert Morris, and Michael Walfish. A world wide web without walls. In *Proceedings of the 6th ACM Workshop on Hot Topics in Networks*, Atlanta, GA, USA, Nov 2007.

[Lampson 1973] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[Livshits 2008] Benjamin Livshits and Weidong Cui. Spectator: Detection and containment of javascript worms. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 335–348, Boston, MA, USA, Jun 2008.

[Lunt 1990] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The seaview security model. *IEEE Transactions on Software Engineering*, 16(6):593–607, 1990.

[McIlroy 1992] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, 1992.

[Miller 2008] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript, 2008. `http://code.google.com/p/google-caja/downloads/list`.

[Myers 1997] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142, Saint-Malo, France, Oct 1997.

[Reis 2006] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 61–74, Seattle, WA, USA, Nov 2006.

[Reis 2007] Charles Reis, Steven D. Gribble, and Henry M. Levy. Architectural principles for safe web programs. In *Proceedings of the 6th ACM Workshop on Hot Topics in Networks*, Atlanta, GA, USA, Nov 2007.

[script.aculo.us 2009] script.aculo.us. Library. `http://script.aculo.us`, Jan 2009.

[Twitter 2009] Twitter. Badge. `http://twitter.com/badges/blogger`, Jan 2009.

[Ubl 2009] Malte Ubl. Xssinterface: Javascript library for secure cross browser javascript messaging. `http://code.google.com/p/xssinterface/`, Jan 2009.

[Vogt 2007] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the 14th ISOC Network and Distributed System Security Symposium*, San Diego, CA, Feb 2007.

[Wang 2007] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 1–16, Stevenson, WA, USA, Oct 2007.

[Yahoo 2009] Yahoo. Pipes. `htpp://pipes.yahoo.com`, Jan 2009.

[Zeldovich 2006] Nickolai B. Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, USA, Nov 2006.