MashupOS: Operating System Abstractions for Client Mashups

Jon Howell
howell@microsoft.com

Collin Jackson
collinj@cs.stanford.edu

Helen J. Wang
helenw@microsoft.com

Xiaofeng Fan
xiaoffan@microsoft.com

***Abstract*—Web browser support has evolved piecemeal to balance the security and interoperability requirements of client-side script services. This evolution has led to an inadequate security model that forces Web applications to choose between security and interoperation. We draw an analogy between Web sites' sharing of browser resources and users' sharing of operating system resources, and use this analogy as a guide to develop protection and communication abstractions in MashupOS: a set of abstractions that isolate mutually-untrusting web services within the browser, while allowing safe forms of communication.**

I. Introduction

Web browsers are becoming the single stop for everyone's computing needs including information access, personal communications, office tasks, and e-commerce. Today's Web applications synthesize the world of data and code, offering rich services through Web browsers and rivaling those of desktop PCs. Browsers have evolved to be a multi-principal operating environment where mutually distrusting Web sites (as principals) interact programmatically in a single page on the client side, sharing the underlying browser resources. Consider a scenario (Figure 1) wherein an HTML file (possibly including scripts) sent from `webmail.com` and an HTML file sent from `stocks.com` run on the client browser. These HTML files are really delegates on behalf of `webmail.com` and `stocks.com`, respectively, using resources on the client to improve the interactivity of the services. In this scenario, the sites are mutually distrusting principals sharing the browser's resources (display, memory, CPU, network access). This resembles the PC operating environment where mutually distrusting users share host resources.

However, unlike PCs that utilize multi-user operating systems for resource sharing, protection, and management, today's browsers do not employ any operating system abstractions, but provide just a limited binary trust model and protection abstractions suitable only for a single principal system: There is either *no trust* across principals through complete isolation or *full trust* through incorporating third party code as libraries. The browser abstraction for the former is FRAME; frames enable interactive (script-enhanced) Web services to occupy neighboring display real estate, but the components

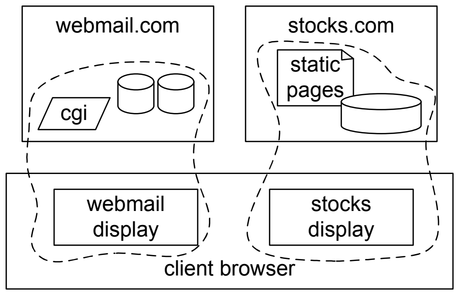

Fig. 1. JavaScript code running on a client browser is really just a distributed component of the Web service that provided the code.

cannot interact. The abstraction for the latter is SCRIPT which allows third-party scripts to be included as library code; the embedded cross-domain scripts enjoy full trust from its includer and can access the includer's data, display, and access to back-end server resources. With these limited existing browser abstractions, Web programmers are forced to make tradeoffs between security and functionality, and often times sacrifice security for functionality. In the scenario above, a web program either segregates HTML content from `webmail.com` and `stocks.com` into separate frames denying any communications or embed their scripts as libraries into a containing page allowing intimate interactions. As we can see, controlled interactions may be desired: If the `stocks.com` server offers a limited Web interface that other servers such as `webmail.com` may access, then the browser should allow similar communication between the corresponding components running on the client. This controlled communication among otherwise isolated client components is not attainable in today's browsers.

In the MashupOS project, we aim to design and build a browser-based multi-principal operating system. In this position paper, we outline our initial explorations on the proper abstractions needed for protection and communciations which to date have received only ad-hoc band-aids and patches. By identifying an appropriate, strong analogy to conventional operating system design, MashupOS draws on decades of wisdom and experience in managing isolation and communication among untrusted principals. While there is no way to show an open architecture to be complete and correct, the MashupOS approach has a solid foundation.

For the rest of the paper, Section II identifies the limitations of the abstractions and security policies implemented in contemporary browsers. Section III sets out the goals of MashupOS. Section IV introduce primitive abstractions which plainly enforce a security model analogous to a multi-principal operating system, providing only spare communication primitives. Section V enhances those primitives with syntactic sugar, providing familiar, simple communication without destroying the security of the primitives. Section VI considers deployment issues, and Section VII relates MashupOS to other proposals. Finally, we conclude in Section VIII.

II. Background

The security policy of current browsers is the result of a patchwork of decisions made by many independent companies and individuals, with a heavy emphasis on avoiding vulnerabilities in legacy sites, rather than providing the best abstractions for the newest sites. To motivate our proposal, we examine the security policies of current browsers and the limitations that they place on Web mashups.

A. Same-Origin Policy

Browsers use cookies as a way to identify and authenticate unique users, and to operate in a way that depends on which user is viewing the page. A cookie is a small, arbitrary piece of data chosen by the Web server and sent to the browser in an HTTP header when responding to a request. On subsequent requests, browsers use HTTP headers to echo back cookies to the server that sent them [8].

In order for cookies to be used as an authentication mechanism and provide the illusion of an isolated session shown in Figure 1, the browser must keep cookies secret from other sites. Thus, cookies are sent only to the same site that set them, a policy known as the *Same-Origin Policy* [6].

B. AJAX

Recently, the *AJAX* programming model has emerged, allowing web services to shift interactive user interface code from the web server to the browser. AJAX stands for Asynchronous JavaScript and XML. Where conventional web pages handle every click with a round-trip to the server, AJAX uses client-side code ("JavaScript") to handle many user interactions, providing interactivity not bounded by network and server performance. Furthermore, when communication with the server is required, that communication occurs asynchronously ("asynchronous XML"), while the client-side code continues to provide interactivity in the meantime.

The Document Object Model (DOM) is an interface that allows scripts to read and modify HTML documents, even documents in other pages or frames. To ensure that web pages cannot circumvent firewalls or hijack the user's authenticated sessions, JavaScript documents loaded from one origin are prevented from getting or setting properties of a document from a different origin. Each browser window, FRAME, or IFRAME (inline frame) is a separate document, and each document is associated with an origin on the basis of URL. Two origins are considered separate if they differ by scheme (http or https), DNS name, or TCP port [12]. For example, frames from `http://amazon.com/` and `http://amazon.co.uk/` cannot access each other's resources because their DNS names differ.

The asynchronous XML communication of AJAX is accomplished using XMLHttpRequest, which can communicate only with the page's origin server. For example, a frame from `http://amazon.com/` cannot issue an XMLHttpRequest to `http://amazon.co.uk/`.

C. Remote Code Inclusion

Web developers often wish to incorporate third-party code libraries. An example is housingmaps.com, which uses the Google Maps code library to visualize Craigslist housing classified ads. Because JavaScript files generally do not have any user-specific or sensitive information, browsers interpret files in this format as public code libraries and allow them to be executed across domains. The code runs with the privileges of the page including it. For example, the `housingmaps.com/index.html` page may contain the markup `<script src='http://google.com/maps.js'> </script>`, which allows `maps.js` to access housingmaps.com's HTML DOM objects, cookies and data through XMLHttpRequest. However, `maps.js` cannot access google.com's resources since the code in `maps.js` is considered to have the origin housingmaps.com rather than google.com in this context.

The existence of remote code inclusion as an alternative to the isolation of the Same-Origin Policy presents web developers with a dilemma: a site must either completely distrust another site and segregate itself through the use of cross-domain frames, or a site can use another site's code as its own, offering full resource access to the remote site.

D. Web Mashups

Web mashups compose data from more than one site, yet the browser prevents such cross-domain communication. XMLHttpRequest cannot retrieve data from another domain, even if the other domain desires it.

Initially, mashup developers worked around these restrictions using a proxy approach: a web portal like MSN.com may, in the back end, compose dozens of web services together into a single web page which is displayed in the browser. Services from different principals can be composed the same way; examples include metacrawlers and news aggregators. This approach unfortunately makes several unnecessary round trips, reducing performance, and the proxy can become a choke point, limiting scalability.

An alternative to proxies is encoding public data in executable JavaScript format (JavaScript Object Notation, or JSON [2]). Using SCRIPT tags, this data can be passed from the provider to the integrator across domain boundaries, eliminating the need for proxies. As a possibly unintended side effect, this technique grants the integrator's privileges to the data provider.

E. Gadget Aggregators

Mutually distrusting network servers communicate with one another using web service APIs, but the Same-Origin Policy offers no equivalent communication among client-side components. As a result, web services that wish to enjoy tight client-side coupling must abandon entirely the isolation afforded by the policy, and instead compose scripts directly using the SCRIPT tag. Scripts composed this way can communicate because all the scripts are treated as belonging to the domain of the enclosing document. Unfortunately, that communication is completely unconstrained. Two examples of compositions follow in which such trust is inappropriate.

Web gadget aggregators such as Google Personalized Homepage page [5] and Windows Live [9] aggregate user-selected interactive content from disparate sources into a single portal page. A *gadget* includes both HTML and JavaScript, and is designed to be included into a gadget aggregator page; it is the client-side of some web service.

Gadget aggregators are security-conscious; third-party gadgets are hosted on a separate domain and IFRAMEs are employed to isolate these gadgets from one another and from the containing page. However, because these IFRAMEs cannot communicate, aggregators also support *inline* gadgets, SCRIPTs inlined directly into the aggregator page. Because inlining requires complete trust, Google's aggregator asks the user what to do: "Inline modules can ... give its author access to information including your Google cookies and preference settings for other modules. Click OK if you trust this module's author."

AOL provides a chat widget, called Web AIM, designed to be integrated into other web services' displays. A web service can interoperate with the widget using an API to add contacts. Ideally, the service should not be allowed arbitrary access to the widget, with which a malicious service might extract the user's list of contacts or even feign a chat session. Absent better abstractions, AOL chooses interoperability, and punts security to the user with a "click OK" dialog box.

III. GOALS

Secure, interoperating client-side service compositions demand new browser abstractions with three properties.

- **Cross-domain protection** prevents code in one domain from compromising the confidentiality or integrity of other domains. To prevent denial of service, domains should receive fair shares of commodity resources such as CPU, network bandwidth, and disk space.
- **Controlled cross-domain communication** allows a service from one domain to interoperate with a service from another, enabling rich composition.
- **Doing minimal violence** to the existing Web API eases adoption of the new abstractions, and the mechanisms must offer acceptable backwards-compatibility behavior.

IV. PRIMITIVE ABSTRACTIONS

Section IV-A identifies the resources MashupOS manages. The SERVICEINSTANCE of Section IV-B isolates principals from each others' resources, and the abstractions of Section IV-C provide a restricted communication model among SERVICEINSTANCEs.

A. Resources

Reusable commodity resources, such as CPU, memory, and network bandwidth, need only be fairly shared to prevent misbehaving principals from denying service to well-behaved principals.

Unique resources, however, require access control abstractions to misbehaving principals from violating confidentiality or integrity. **Persistent storage** in the browser is much simpler than conventional OS file systems; therefore, rather than extend the OS analogy to share persistent state among principles, MashupOS maintains the isolated persistent storage model of today's browsers. Likewise, MashupOS offers no shared **memory** between principals, so no memory access-control abstraction is necessary.

In MashupOS, the **display** is an access-controlled resource in the same sense that the X11 window server enables mutually-distrusting clients to share access to a common display resource.

Likewise, **network access** from client code is subject to access controls that mimic those enforced on accesses originating at the principal's web server, following the intuition of Figure 1. Indeed, communication between client components is treated as network communication, and subject to the same access controls.

B. The ServiceInstance Isolation Primitive

We propose a primitive abstraction called a SERVICEINSTANCE, analogous to an operating system process. A new browser window is initially associated with a single SERVICEINSTANCE, which contains the page's Document Object Model (DOM) structure and the ephemeral state (variables) associated with the code on the page. The SERVICEINSTANCE is also associated with a single principal, defined as the Same-Origin Policy domain associated with the document loaded into the page.

Just as in today's HTML, if a SERVICEINSTANCE uses a SCRIPT tag to incorporate a trusted library by reference, that library runs as the principal associated with the SERVICEINSTANCE, regardless of the location from which the script was fetched. The same effect can be achieved if the SERVICEINSTANCE principal's server fetches the script and inlines it into the SERVICEINSTANCE's HTML: in both cases, the SERVICEINSTANCE is owned entirely by the principal that served the outer HTML, and the inlined script is completely trusted to run on behalf of that principal.

We introduce a new HTML element `<FRIV src="page2.html">` that crosses the security isolation of a FRAME with the layout and communications benefits of a DIV. The FRIV has three effects. First, it allocates a subregion of the outer display region. Second, it creates a new SERVICEINSTANCE. Third, it populates the DOM of the subregion by loading the referenced SRC document.

The DOM that represents the content of the subregion is isolated inside the new SERVICEINSTANCE. Script code in the inner SERVICEINSTANCE cannot reach the DOM of the outer SERVICEINSTANCE by traversing pointers (parent pointers, callbacks, or anything else), nor vice versa. Likewise, the ephemeral state of the scripts loaded into the inner SERVICEINSTANCE are isolated: the inner SERVICEINSTANCE cannot hold a reference to objects in the outer SERVICEINSTANCE, nor vice versa.

The outer SERVICEINSTANCE (that created the FRIV) and the inner SERVICEINSTANCE divide responsibility over the display resource. The outer SERVICEINSTANCE is responsible for all of its display other than the contents of the FRIV display region, and the inner SERVICEINSTANCE is responsible for the contents of the FRIV. The outer SERVICEINSTANCE's DOM represents the delegated display region with a "top-half FRIV" object: the outer SERVICEINSTANCE can adjust the display area of the FRIV, but cannot see the composite DOM below that point. Likewise, the inner SERVICEINSTANCE's root

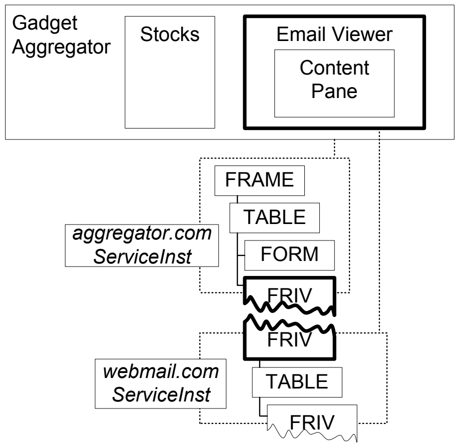

Fig. 2. The diagram on top shows the subdivision of a display surface into regions. Dotted boxes on the below represent the SERVICEINSTANCEs that contain the DOM elements and other private data associated with each display region. The two SERVICEINSTANCEs shown adjoin by a FRIV object, highlighted by a bold line. Each SERVICEINSTANCE can only access its "half" of the FRIV object. The FRIV represents the subdivision of display space (shown by the bold boundary in the top diagram) and provides an explicit, data-only communication channel between the SERVICEINSTANCEs.

DOM object is a "bottom-half FRIV" object: the inner SERVICEINSTANCE can read the size of the allocated region, and populate it with DOM elements, but cannot see the composite DOM elements above the FRIV boundary.

FRIV can be used recursively: a FRIV may contain another FRIV. For example, a gadget aggregator (in a FRAME) may contain a mail-reading gadget (in a FRIV), which may recursively contain a content-viewing pane (in another FRIV).

C. Communication among SERVICEINSTANCE*s*

The FRIV is the DOM object that connects two SERVICEINSTANCEs: The outer SERVICEINSTANCE delegates control over part of its display to the inner SERVICEINSTANCE. Each SERVICEINSTANCE is a represention on the client of some web site; sites may reasonably wish to communicate in some limited (mutually-distrusting) fashion between SERVICEINSTANCEs. Therefore, the FRIV includes an explicit communication channel between the SERVICEINSTANCEs it joins.

The inner SERVICEINSTANCE can write messages into the bottom-half FRIV, and the outer SERVICEINSTANCE can register a callback with the top-half FRIV to receive such messages. The messages are data-only, one-way messages, and therefore have semantics equivalent to discrete network messages between the servers that are responsible for the SERVICEINSTANCE content. In particular, data-only messages mean that a SERVICEINSTANCE cannot hang itself by giving away a pointer to its DOM or internal state. Our choice of an asynchronous communication abstraction ensures that ServiceInstances can communicate without failure-coupling to one another; AJAX-style asynchronous RPC is easy to build over asynchronous messages.

The primitive syntax for explicit communication channels emphasize the simplicity of the channel, in particular its equivalence to a network channel: Each half-friv object contains an `registerReceiveHandler(handler)` method, which registers a JavaScript function to receive data, and a `send(data)` method, which sends data to the handler on the other end of the channel. The receiver also learns the identity of the principal that sent the message.

The MashupOS implementation allows messages to contain only value types (including compounds). Because messages may *not* transmit a pointer to data in the sending ServiceInstance, they behave just as a network message between the sites on behalf of which the ServiceInstances run.

D. Commodity Resource Isolation

Given the ServiceInstance isolation abstraction, we can apply conventional techniques to fairly share commodity resources such as CPU, memory, and network bandwidth. This keeps a resource-hog from breaking other services.

For pedagogy, we described each ServiceInstance as associated with exactly one display region. In practice, we expect to generalize the ServiceInstance to manage zero or multiple display regions, providing the analog of background processes and multi-window applications.

E. Null-Principal ServiceInstances

We include as an optional enhancement the ability for a principal to create a ServiceInstance with a null principal. Such a ServiceInstance receives the usual fair share of commodity resources, and may also draw into its friv and communicate with its creator via its friv. Because the ServiceInstance has no principal, it cannot make XMLHttpRequests; any JSONRequests carry the null principal as their origin. Cookies, which are associated with principals, are unavailable to the null-principal ServiceInstance. The email gadget display can use a null-principal ServiceInstance to render untrusted email content fetched from `webmail.com` without giving that content access to the principal's resources.

V. Syntactic Sugar Abstractions

The friv primitive described in Section IV is like today's cross-domain FRAME, plus commodity resource isolation and a message-based communication channel. Developers, however, prefer `DIV`s to `FRAME`s for two reasons. First, `DIV`s better negotiate display real estate to accomodate documents of varying size. Second, communication between the gadget charged with implementing a DIV and the containing document is as simple as method invocation. In this section, we enhance the friv to exhibit these preferred behaviors. These enhancements do not weaken the primitive model; indeed, it is possible to implement them as syntactic sugar over the primitive model.

One advantage of using a DIV rather than a FRAME to contain unknown content is that DIVs automatically resize to fit their content, whereas FRAMEs force the user to drag through the content with a scrollbar. A friv is rendered like a frame by the browser, to prevent the inner gadget from using absolute positioning to display elements outside the friv's boundary. However, by passing width and height messages between the outer and inner documents, our friv automatically resizes itself to fit its content. The parent page can override this behavior using stylesheets to specify a fixed or maximum size for the friv, or by explicitly intercepting the messages with its own handler. The page inside the friv must activate the automatic resizing feature explicitly by calling the `exportSize` function; this consent prevents an attacker web site from determining the number of bytes in a confidential document from another domain by measuring the friv's size.

The sugared friv also provides the illusion of direct communication via function calls; in the operating systems analogy, we provide an asynchronous remote procedure call abstraction. This abstraction retains the familiar syntax of contemporary mashups.

Suppose Alice hosts a mashup that uses third-party map software provided by Bob. Alice.com includes Bob's map software by rendering a friv that points to Bob.com. In order to allow Alice to re-center the map to a particular location, Bob defines a JavaScript method `setCenter(latitude, longitude)` and allows Alice to call it by using the syntax `parent.export(setCenter)`, where parent is the bottom-half friv at the root of Bob's DOM. Alice can now call Bob's `setCenter` method by invoking `exportedMethods.setCenter()` on the top-half friv that adjoins to Bob's ServiceInstance.

Alice can also export methods to Bob. For example, Alice might want to be notified when the user clicks on a point on the map. Alice can define the function `onclick(latitude, longitude)` and allow Bob to call it by calling the `export(onclick)` method of the top-half friv adjoining Bob's ServiceInstance. Bob can now call Alice's onclick method by invoking `parent.exportedMethods.onclick`.

The simple examples shown above do not involve return values; we also provide a capability to make asychronous (using an additional JavaScript callback argument) calls to functions with return values. We also propose a synchronous interface, although synchronous RPC failure-couples the caller to the callee.

The FRIV tag is designed for web mashups where the caller and callee only partially trust each other. It is important to note that this syntactic sugar does not violate the data-only limitation of the communication channel (Section IV-C). If the participants trust one another enough (arguably completely) to pass code or references to DOM objects, then an inline `SCRIPT` tag suffices (in the operating systems analogy, the caller links to the third-party library directly).

One exception to the data-only rule is that a communication endpoint itself may be passed as an argument in a cross-SERVICEINSTANCE message; analogous in the OS world to passing a file descriptor through an IPC channel. This feature enables a gadget aggregator to directly connect two of its siblings. We have yet to solidify the details of this proposal.

VI. Proposed Implementation

For a proposal like MashupOS to be feasible, it must be implementable. We plan for future work a reference implementation of the MashupOS abstractions. Here, we consider alternative implementation approaches.

A. Isolation

Isolation of separate FRIVs, including commodity resource allocation, can be provided natively in the browser or a plugin, based on JavaScript type safety. Alternatively, isolation could be implemented using processes [10], virtual machines [1], or dynamic code transformation.

Code transformation offers an interesting alternative to the browser-upgrade deployment path. BrowserShield [11] is a framework that dynamically rewrites an HTML-and-JavaScript page to obey a given policy. One killer application of BrowserShield was to protect browser vulnerabilities by catching even dynamically-generated exploit code. A BrowserShield implementation of MashupOS would enforce isolation by preventing code from from following references across FRIV boundaries in the display DOM. Dynamic rewriting can incur significant performance overhead. Its advantage is that it can be deployed remotely, e.g. by a gadget aggregator site, to enhance legacy browsers with the MashupOS abstractions.

B. Incremental Deployment

Until all browsers support MashupOS, websites must handle users with legacy browsers that do not understand the FRIV tag, by providing a safe, user-friendly fallback behavior. To that end, we borrow a trick from IFRAME: FRIV-aware browsers shall ignore the contents of a FRIV tag. Legacy browsers ignore the unsupported FRIV tag, and proceed to render its contents normally. Thus, a web developer can place inside the FRIV tag a cross-domain IFRAME, ensuring that the referenced gadget is safely displayed, regardless of browser support for FRIV. Alternatively, the web developer may use the FRIV content to link to a FRIV-enabled browser upgrade or plugin.

To ensure that MashupOS web pages are considered valid XHTML, the FRIV tag is part of a custom XML namespace until it is someday adopted as part of the HTML standard. Thus, technically a FRIV should be created with the slightly longer syntax <`mashupos:friv src='page2.html' xmlns:mashupos= 'http://research.microsoft.com/mashupos/'`>.

Instead of introducing a new tag, an alternate approach would be to reuse an existing tag, such as IFRAME, SCRIPT, or OBJECT. Internet Explorer and Opera adopted this approach with the `security=restricted` attribute of IFRAME, which allows a containing page to render a frame with JavaScript and other active content disabled. Unlike the FRIV tag, the security-restricted IFRAME tag does not fail safely: When encountered by a browser such as Firefox that is not aware of the `security` property, the active content is allowed to execute.

VII. Related Work

Recently, a new wave of "web operating systems" [3] (e.g., YouOS [13]) have emerged. These sites present a traditional desktop user interface, complete with a window manager. The applications run natively in JavaScript. All are hosted on the same domain as the web desktop, and thus have unlimited access to one another. This lack of isolation, comparable to the 1995 PC desktop, requires the user to completely trust every application that is run.

Our earlier work on Subspace [7] provided a cross-domain communication mechanism that is designed to run on current browsers without any additional plug-ins or client-side changes. Subspace uses the browser's existing `document.domain` property to communicate with isolated subdomains, which are in turn used to draw in scripts from other domains. However, Subspace requires significant work on the part of the web developer to use correctly, and does not provide the resource constraint options of FRIV. We believe browsers should provide built-in cross-domain interaction primitives.

Crockford recently proposed the JSONRequest [2] communication mechanism for asynchronous cross-domain data retrieval from a remote server. The proposal was motivated by scenarios where the cross-domain SCRIPT tag was being used to execute code when only data was really required. The JSONRequest's usage is similar to XMLHttpRequest, but it is not constrained by the Same-Origin Policy. Lifting these same-origin restrictions is safe because cookies are not sent, because the request includes a header indicating the source of the request, and because the server's reply must indicate the server is aware of the protocol and hence its security implications. JSONRequest transmits data in the JSON format, but its security applies equally to XML or other formats. A cross-domain JSONRequest is the client-side equivalent to a cross-server TCP request, and thus JSONRequest complements MashupOS well. Alternatively, one can simulate JSONRequest in MashupOS by creating a FRIV that communicates with its home domain and then passes the received data back to the outer SERVICEINSTANCE.

Crockford also identified the security limitations that affect today's cross-domain mashups. In response, he proposed a new HTML tag, the MODULE tag, to partition a page into a collection of modules [2]. A module groups DOM elements and scripts into an isolated environment; socket-like communications are allowed between the inner module and the outer module. To isolate the module from the origin server, modules may not make network requests. Thus, modules are equivalent to null-principal FRIVs.

Cross-document messages [4] are a proposed browser standard that would allow cross-domain frames to send string messages to each other on the client side. Cross-document messages are thus similar to the messaging capabilities of full-principal FRIVs. They are implemented in the Opera browser. We expect that with better data type support, automatic layout capabilities, and other syntactic sugar provided by FRIV, wider deployment and use of this messaging paradigm can be achieved.

VIII. CONCLUDING REMARKS

Client mashups enable a new generation of user-friendly and feature-rich web applications. While mashups turn the browser into a multi-user system with mutually distrusting domains as users, today's browsers offer web developers insufficient abstractions for integrating content from different domains: either cross-domain isolation with no communication or uncontrolled communication with no isolation. MashupOS applies operating system principles to bridge this gap. We introduce the SERVICEINSTANCE as the unit of resource isolation, data-only message-based communication between SERVICEINSTANCEs, and the FRIV as the abstraction of display sharing. Invoking well-understood operating system principles promises to provide a stable security foundation to replace today's teetering *de facto* abstractions.

REFERENCES

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

[2] D. Crockford. JSON. `http://www.json.org/`.

[3] Big WebOS roundup - 10 online operating systems reviewed. `http://franticindustries.com/blog/2006/12/21/big-webos-roundup-10-onlin%e-operating-systems-reviewed/`.

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

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

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

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

[8] D. Kristol and L. Montulli. HTTP State Management Mechanism. IETF RFC 2109, February 1997.

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

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

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

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

[13] YouOS. `http://www.youos.com/`.