Secure Cooperative Sharing of JavaScript, Browser, and Physical Resources

Leo A. Meyerovich, David Zhu
{lmeyerov, yuzhu}@eecs.berkeley.edu
University of California, Berkeley

Benjamin Livshits
livshits@microsoft.com
Microsoft Research

ABSTRACT

For better application-level controls on mashups, we advocate extending the Single Origin Policy and associated primitives to support a cooperative model that allows applications to express explicit sharing policies over browser, Javascript, and physical resources.

First, we introduce an isolation model for content loading that is more complete than those of surveyed browser proposals. Second, we present new primitives to enable an application to secure its use of untrusted content by delegating browser, JavaScript, and physical resources in a fine-grained and reliable manner. Finally, essential to adoption, we propose an architecture based on designs for related abstractions with low performance and implementation costs.

1. INTRODUCTION

As web content evolves from static documents to server-generated components that interact with third party libraries, user-generated content, and even third-party applications, trust relationships have become more nuanced than shared nothing (different origin) or shared everything (same origin). Many policies are being proposed, but they are often too static or coarse. Worse, they are enforced with either hard-coded browser instrumentation or indirect techniques like page rewriting and script wrapping. In summary, browser support for security is efficient and reliable, but browser policies do not address appliciation-specific concerns. While there have been significant advances in isolating principals in browsers, efficient and reliable abstractions and mechanisms to safely share resources between otherwise isolated principals are less clear.

In this paper, we attempt to bridge this gap by proposing a cooperative model that allows fine-grained sharing between different-origin principals of browser, JavaScript, and physical resources subject to scriptable policies. To motivate our proposal, consider a site like Facebook that enable users to embed third-party applications and share rich content with each other. Figure 1 shows how Facebook might use a few new isolation and sharing primitives we will describe in Section 3 to achieve application control of these resources:

```
1. var toggle = true;
2. delegateNetwork(gadget, "http://gadget.com",
3.  function () { if (toggle) return true; });
4. function giveData () {
5.  toggle = false;
6.  return "profile data"; }
7. around(gadget, giveData,
8.  function (proceed) { return proceed(); });
9. ... <CoFrame src=http://gadget.com/page id=gadget
10.       passthrough="html css js"
11.       delegate=".1 cpu"/> ...
```

Figure 1: Facebook giving a gadget limited network, function, and CPU time access.

Controlling browser resources (lines 1-6, 9-10). If a gadget is to be empowered to visualize privileged user profile information, a simple policy to keep the data confidential would be to force the gadget to pre-load all external resources, and then, before providing it privileged data, disable the gadget's network access. In the current model, gadgets get access to various browser resources such as cookies, network access automatically, making these types of policies difficult or impossible to implement. In our new proposal, Lines 9-10 load the gadget with no facebook-origin nor gadget-origin network access and lines 1-6 turn gadget-origin network access on and off. Note that access must be delegated and is disabled by default. The controlling principal, in this case Facebook, defines the policy.

Other gadget-origin browser resources of interest include the ability to execute JavaScript as in ConScript [11], create and modify DOM elements as with METs [4], and use persistent storage. Furthermore, while ConScript enables control of the IE8 JS interpreter's access to COM (e.g., browser) objects, it could not control many security-critical calls, such as inter-native component ones experiment upon within the OP Browser [6]. We advocate subjecting these to application control.

Controlling script resources (lines 4-7). In the current browser model, different origin frames share no Javascript objects between one another and are only guaranteed secure communication of immutable structures like strings. The inability to share objects like records and functions restricts the programming model. As web applications grow, so does the need for more flexible sharing scheme. We advocate associating every JavaScript heap object with a principal as is already common in browsers [5]. By default, every such value is inaccessible to other principals. We propose, using the cross-principal advice function `around` in line 7, a principal may provide access of one particular heap object to a different principal: even if there are other heap objects connected to the shared object on the reference graph, cross-principal access to them is still denied as they have not been explicitly shared.

Controlling physical resources (line 11). Browser responsiveness and other forms of quality of service are increasingly critical. However, current browser-supported policies allow malicious components to monopolize resources. First, consider CPU scheduling in recent secure browsers, where, notably, Chrome provides fair scheduling between windows and Gazelle between frames. Despite the goal of performance isolation, in both cases, deviant content may still dilute CPU time of other principals by creating new windows or frames, respectively. Second, compute-intensive web applications and certain functions of the browser may employ highly optimized code that is sensitive to changes in the underlying hardware usage, such as cache usage, task migration etc. [12] Thus, in both adverserial and benign scenarios, having knowledge and control of physical resource can improve the perceived performance of browsers and the applications.

As shown in line 11 of Figure 1, Facebook can set a bound on the resources available to the gadget. This demonstrates an application may customize content's use of physical resource to its need instead of adopting a static browser-defined policy. Similar to our browser and JavaScript primitives, such sharing may be dynamic and even scriptable.

In the following sections, we first argue for application-level control of these resources and browser support to reliably and efficiently do so. We state our goals and examine the current system to see why they are not satisfactory (Section 2). Next, we present our main contribution of this paper, a set of more complete isolation and finer-grained sharing primitives for 1) browser, 2) JavaScript and 3) physical resources (Section 3). Finally, we examine how our approach might be efficiently implemented. (Section 4) and reflect upon strengths and challenges of our approach.

2. TRUST AND PRIVILEGE MODEL

We are interested in *browser support* of *application-specific policies* and in this section we make a case for both. We present additional requirements of the resource policies and evaluate current solutions to see where they fall short.

2.1 Principles

Systems should observe the following principles:

Application-specific policies. Given the diversity of web applications, one-size-fit-all policies are no longer sufficient. Browsers need to provide primitives to the applications to customize policies, rather than forcing every application to adopt the same policy as it is done today with SOP. More subtly, achieving the separation of policies from mechanisms would decouple application security from the unreliable browser upgrade cycle and enable more rapid iteration over policy design.

Browser support. Policy enforcement should primarily rely upon browser-provided support. This has several advantages over other browser independent techniques such as script rewriting and wrapping. For example, ConScript [11] shows benchmarks that perform 2.7 times better. In addition, rewriting and wrapping are fragile, leading to excessive privilege escalation as shown in [5] and [10]. Finally, new browser features and even modifications often break the assumptions made by the rewriting and wrapping tools, leading to inconsistent policies and therefore security breach.

Cooperative sharing. The proposed resource management scheme should match the structure of the browser, which follows a hierarchical manner: a browser delegates resources to the top level containers (e.g., a window or tab), which then specifies the resource policy of its embedded entities. We must protect gadget use of gadget-origin resources as well: a container should be able to toggle a gadget's access to them, but unable to otherwise tamper with toggled (gadget-origin) resource interactions.

The resource policies should be fine-grained, deep and default-deny. First, the granularity of resource policy should match the programming model. For example, applications in JavaScript use functions and objects to model domain concepts, so application security controls should be on more than just strings, such as is achieved by object capability languages. [14] Second, we advocate *deep advice* policies that associate protection logic with an object and its interactions, rather than on its access path. This is more reliable than indirect (alias-based) techniques like wrapping or rewriting [11] by providing direct mediation. Finally, following the principle of least authority, we should not allow loaded gadgets to access any resource that has not been explicitly shared by the container. We argue that when one JavaScript object is passed to an untrusted principal, other objects referenced by it (the reference graph) should, by default, still be inaccessible by the untrusted principal even though they are accessible to the owning principal. Such control is expressible, for example, with *object views* [13],

2.2 Compliance of Modern Systems

In Figure 2, we compare sharing support of various standard and proposed browser primitives according to our guidelines (default deny, granularity/directness/scriptability, tampering). The surveyed tools, in order, are string message passing frames in current browsers, Gazelle service and null instances [16] (more powerful than Google Chrome frames), OMash cross-origin access control lists (with reference passing) [3], Google Caja with same-frame (rewritten) gadgets [15], object views using a a serialization protocol for passing remote object access between different-origin frames [13] over the postMessage string-passing primitive, and ConScript [11] for same-frame JavaScript advice. The bottom row is an idealized approach (to be proposed in Section 3) that is at least as good as the rest in every category.

Consider a container sharing container-origin resources with a gadget. Gadgets typically cannot access container JavaScript values by default: ConScript is an exception as it uses same-origin script tags and selectively disables (instead of enabling) access. More concerning is unit of control. Almost all proposals that support sharing of values beyond strings share references rather than individual values: passing one reference implicitly passes any others referenced by the value (the reference graph), violating the principle of least authority (POLA). Figure 2 differentiates betweens between unrestricted sharing of entire reference graphs (`ref`) and the directly referenced heap object (`value`).

Browser resource sharing has similar challenges as JavaScript resource sharing, except reference passing has the additional challenge that DOM APIs do not directly reflect deep functionality. For example, there is no accessible value

	Gadget Access of Container-Origin Resources						Gadget Access of Gadget-Origin Resources				
	JavaScript		Browser		Physical		JavaScript	Browser		Physical	
Mechanism	d. deny	control	d. deny	control	d. deny	control	untampered	d. deny	control	d. deny	control
frame	✓	string	✓	string	✗	✗	✓	✗	✗	✗	✗
serviceinstance	✓	string	✓	string	✗[f]	✗	✓	✗	✗	✗[f]	✗
nullinstance	✓	string	✓	string	✗[f]	✗	✓	✓	✗	✗[f]	✗
omash	✓	ref	✓	ind. ref	✗	✗	✓	✗	✗	✗	✗
caja$_{\text{same-frame}}$	✓	ref	✓	ind. ref	✗[c]	✗[c]	✗	✓	✗	✗[c]	✗[c]
caja$_{\text{diff-frame}}$	✓	ref	✓	ind. ref	✗	✗	frame	✓	all/none	✗	✗
object views	✓	value	✓	ind. val	✗	✗	✓	✗	✗	✗	✗
conscript	✗[b]	value	✗[b]	value	✗	✗	✗	✗	all/none	✓	✗
coframe$_{\textbf{(ideal)}}$	✓	**value**	✓	**value**	✓	**value**	✓	✓	**value**	✓	**value**

[b] Opt-in (e.g., blacklist). [c] Same-frame JavaScript CPU control in Web Sandbox [f] Gadgets are fairly scheduled with the container, giving excess privilege
[ref] Sharing a JavaScript value passes a reference graph [val] Sharing a JavaScript value only enables direct access to just that value
[ind] Security-critical functions are not exposed to direct JavaScript control

Figure 2: Comparison of browser-side application-specific cooperative sharing approaches.

representing container-origin network calls. To control such functionality, tools must approximate access paths, such as controlling XMLHttpRequest and image loading to control vectors for network access, and instrument these. Such an approach – applicational-level *taming* – is fragile because it is unreasonably indirect (`ind` in the table) . Finally, the only support for physical resources is the fair scheduling of Gazelle, which still allows resource exhaustion and does not support delegation, and unreliable timeouts inserted within JavaScript in Microsoft Web Sandbox (a project similar to Caja).

Finally, consider container control of gadget access to delegated gadget-origin resources. For example, a container may want to control its gadget's access to gadget-origin cookies. First, the gadget must be free of container tampering, such as a container reading the value of a privileged gadget-origin cookie field: this is correctly prevented in most proposals, with the notable exceptions of same-frame content in Caja and ConScript that provides excessive control to the container. Another problem is that gadget-origin browser resources are generally enabled by default, and when they are not (Caja and Gazelle), access controls are all-or-nothing. The lack of gadget-origin physical resource sharing is similar to the lack of container-origin resource sharing.

The remainder of the paper proposes the design (Section 3) and implementation (Section 4) of primitives that address the above gaps in access control.

3. ISOLATION & SHARING PRIMITIVES

We revisit our motivating example in Section 1 to explain our new isolated loading and resource sharing primitives. After, we discuss the usability of our proposed primitives.

For simplicity, we assume container and gadget are of different origin and thus interchange *principal* and *origin.*

3.1 Initial Isolation

In Figure 1, we initially isolate the gadget using our new abstraction of a `CoFrame`. A different origin `CoFrame`, by default, cannot access its container's JavaScript, browser, nor physical resources. Similarly, it does not have ambient access to same-origin JavaScript or browser resources. Instead, it must rely upon the container to explicitly grant such resources. In contrast, while a different origin `frame` does not have access to container physical and browser resources, it has undesirable ambient access to the containers physical resources and full access to gadget browser and physical resources.

Various same-origin `CoFrame` instances may have distinct privilege levels, such as only one having the ability to communicate with the outside world. If they can communicate, they can collude. The ability to communicate is therefore disabled by default: a communication channel must be explicitly provided by the container.

3.2 Granting Access to Browser Resources

By default, the gadget does not have *gadget-origin* network access nor *facebook-origin* network access. Line 2 of the policy is executed whenever *gadget-origin* network access is attempted by the gadget. By default, it is disabled. Note that Facebook's tampering with the call, such as by trying to steal a login password, is prevented by default. In contrast, similar to the next example of controlling *facebook-origin* JS value access, Facebook can tamper with *facebook-origin* network access: to force a gadget to choose between tamperable *facebook-origin* network access or no network access at all (due to insufficient privileges), Facebook would not instruct the frame loader to enable the *gadget-origin* network call and instead only delegate it the *facebook-origin* one.

Browser resources to be shared include the foreign function interface exposed for handling the DOM. [11] However, this is often indirect. Just as we advocate the ability to control script injection, [11] which is not directly represented in the current DOM nor JS libraries, there are other security-critical end-points in the browser, such as file and network IO. The Content Security Policy and OP Browser [6] projects highlight additional resources of interest even if they provide little to no control over them.

3.3 Sharing JavaScript Resources

A `CoFrame` gadget has references to its container and *gadget-origin* JavaScript objects, but their usage is denied by default. First, this is because it does not have the ability to execute code, which is a *gadget-origin* browser resource to be toggled by the container. Once granted such access, we focus on a primitive for enabling gadget access to *container-origin* JavaScript values.

We use our notion of *object views* [13] to share JS values (as opposed to just strings or difficult-to-manage chunks of reference graphs). A container may provide access for particular JS values to particular origins, as in line 7 of the ex-

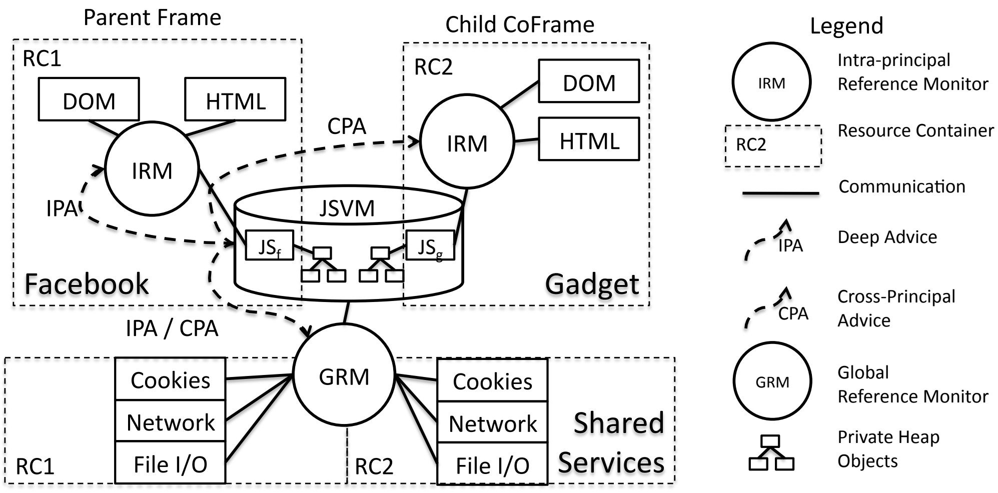

Figure 3: Proposed Architectural Design.

ample. Any interaction with the value, like calling, reading, and writing, triggers the interpositioning code. For POLA, cross-origin access to a value triggers a default-deny policy: a policy must be explicitly set to enable it.

To protect the gadget from its container, the gadget's JavaScript values are also default-deny for other origins: unlike most proposals, a gadget would also have to explicitly share a JavaScript value with its container. Furthermore, advice is double-sided: beyond just explicitly exporting its JavaScript values, a principal must import untrusted ones. Principals must there explicitly enable what they deem to be safe interaction patterns with untrusted objects.

3.4 Physical Resource Policy

Access to physical resources by a `CoFrame` are also subject to default-deny policies. Otherwise, both the container and the entire system is open to denial of service attacks. Physical resources of interest include CPU time, memory and offline storage consumption, network bandwidth, and, increasingly important on mobile devices, power and energy.

By default, a `CoFrame` receives no access to the physical resource, regardless of its origin. Using high-level abstractions such as lottery tickets, a container may transfer its resources to the `CoFrame` as demonstrated in line 12 of the example. We also allow the container to revoke the resources later (such as changing the attribute on line 11), enabling the dynamic adjustment of resource allocation based on application need and user focus.

3.5 Usability

We argue that our primitives provide a usable interface for both directly specifying policies and when used in conjunction with other tools.

Our primitives are intuitive for a web programmer as they are similar to existing JavaScript primitives. For example, JavaScript already supports advice for setting and getting fields of objects; we extend this ability to also apply advice to invocations of a function object and make it sensitive to the principal performing the action. Similarly, the introduction of principals subsumes points of introduction under the current Same Origin Policy – CoFrames are an alternative to different-origin frames – and provides an analogous extension to CoScripts instead of (insecure) different-origin scripts. Thus, our trust model and associated primitives for increasing privilege are consistent with the existing and widely-deployed browser programming model.

Finally, our primitives are useful even when not directly invoked by application developers. Modern applications employ a variety of languages and frameworks. Even when a developer might not directly use our mechanisms, their tools might. For example, we found a JavaScript function advice primitive to be useful for securing the output of a C#-to-JavaScript compiler [11]. Similarly, frameworks like Caja for widgets, AdSafe for advertisements, and blog management systems put significant effort in finding vectors for code injection: basic control of browser resources would make these attempts more reliable. Recent proposals have suggested that application-specific policies can be generated without requiring the author to write code. For example, a static [7] or dynamic analysis [11] can be used to determine acceptable interaction patterns which a developer then simply audits. Given the flexibility of JavaScript, reliabile enforcement in these intrusion detection systems is non-trivial. To achieve reliability, Guha et al advocated performing the intrusion detection checks with a serverside proxy, and therefore at the cost of granularity and performance. Our primitives would enable such tools in terms of reliability and performance without requiring application developers to code policies.

4. ARCHITECTURE

To enable efficient and reliable enforcement of policies like the above, we propose several simple architectural and scripting engine changes. Figure 3 captures three central elements of our design: protection, advice mechanism and cooperative resource management. Each rectangle represents a protection boundary that needs to be enforced. Protections for the shared services, DOM, HTML are achieved through process boundaries, while scripts are isolated based on language safety properties and relies on the JavaScript VM. We stress the choice of isolation mechanisms is independent from the other aspects of the design that follows; we only describe one design. For example, we segregate the reference monitor for local resources and global shared browser resources to limit the impact of a compromised reference monitor.

Cross-principal JavaScript advice. Every JavaScript object may be tagged with an origin so the use of values can be checked for advice (which are just JavaScript functions). In ConScript, we found that checking for advice in the typical case of having none (for calling functions) had an imperceptible cost in Internet Explorer 8's interpreter. Barth et al. have shown that polymorphic inline caches may be reused to efficiently check same-origin access in method-based JITs [2] and we expect these checks can often be compiled away entirely in tracing JITs. When advice is enabled, meaning a policy that requires computation, we expect similar results: micro-benchmark overhead for running an empty advice function was 2-3% in ConScript. Further costs are questions of policy.

Deep browser advice. In ConScript, we provided deep advice to the call to receive text and convert it into executable code, and the foreign function interface (e.g., DOM calls), with uses such as detecting cross-site scripting attacks. There are other components of interest as well for which this is a reliable and efficient approach. The OP Browser, as part of its design, separated core browser components and verified a model of the interactions between them. Furthermore, it routed all inter-component communication through a browser kernel, enabling it to experiment with a variety of browser security policies. If these controls were to be exposed to the application level, more direct reasoning about interactions would be possible. Depending on how components are implemented (native code with heavy processes, managed code, etc.), there are different concerns such as memory footprint and message-passing overhead. However, we believe advances in operating systems, compiler technology, and hardware may prove beneficial here: for example, the OP Browser has found that their partitioning has actually sped up their performance.

Cooperative physical resource management. Cooperative resource allocation enables application's to prevent attacks like DOS. Our interest in better physical resource control also stems from the design of our parallel web browser [8] as much of the time in browsers is spent in native libraries like CSS selector matching. Tuned algorithm approaches [12], reminiscent of typical HPC techniques like tiling, give magnitudes of improvements, so having better control would yield to better speedups. Finally, effective QOS management is not possible without accurate accounting of resource usage. To achieve more accurate accounting of resource consumption, we are advocating concepts similar to Resource Container [1]. Shared services and the JavaScript VM can take on the resource container of the corresponding principal as needed. We outline how an operating system may support many of these ideas in our experimental OS [9].

5. CONCLUSION

Modern browser and web applications are becoming increasingly complex. We propose a design of a few simple browser supported primitives that allows applications to customize its sharing policies of Javascript, browser and physical resources in a more reliable and efficient manner. In particular, we 1) introduce cross-principal advice for JavaScript and propose exposing 2) browser component interfaces and 3) physical resources to such control. We argue that such controls are well-aligned with already successful web programming model and, based on related advances in compilers and operating systems, may be efficiently supported without significant rewriting of browsers. We believe such support is an effective path towards more secure and performant web applications.

6. REFERENCES

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.

[2] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proceedings of the USENIX Security Symposium.*

[3] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 99–108, New York, NY, USA, 2008. ACM.

[4] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, 2007.

[5] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure javascript subsets. In *Proc. of Network and Distributed System Security Symposium, 2010.*

[6] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, 2008.

[7] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for AJAX intrusion detection. In *Proceedings of the International Conference on World Wide Web*, pages 561–570, 2009.

[8] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser. In *Proceedings of the Workshop on Hot Topics in Parallelism*, March 2009.

[9] K. Klues, B. Rhoden, D. Zhu, A. Waterman, and E. Brewer. Processes and resource management in a scalable many-core os. In *HotPar '10: Proceedings of the 2nd Workshop on Hot Topics in Parallelism*, 2010.

[10] S. Maffeis, J. Mitchell, and A. Taly. Run-time enforcement of secure javascript subsets. In *Proc of W2SP'09*. IEEE, 2009.

[11] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[12] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proceedings of the 2010 World Wide Web Conference*, 2010.

[13] L. A. Meyerovich, A. P. Felt, and M. S. Miller. Object views: Fine-grained sharing in browsers. In *Proceedings of the International Conference on World Wide Web*, 2010.

[14] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, 2006.

[15] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript, October 2007. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf.

[16] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.