2010 IEEE Symposium on Security and Privacy

Object Capabilities and Isolation of Untrusted Web Applications

Sergio Maffeis
Imperial College London
maffeis@doc.ic.ac.uk

John C. Mitchell
Stanford University
mitchell@cs.stanford.edu

Ankur Taly
Stanford University
ataly@stanford.edu

***Abstract*—A growing number of current web sites combine active content (applications) from untrusted sources, as in so-called mashups. The object-capability model provides an appealing approach for isolating untrusted content: if separate applications are provided disjoint capabilities, a sound object-capability framework should prevent untrusted applications from interfering with each other, without preventing interaction with the user or the hosting page. In developing language-based foundations for isolation proofs based on object-capability concepts, we identify a more general notion of authority safety that also implies resource isolation. After proving that capability safety implies authority safety, we show the applicability of our framework for a specific class of mashups. In addition to proving that a JavaScript subset based on Google Caja is capability safe, we prove that a more expressive subset of JavaScript is authority safe, even though it is not based on the object-capability model.**

***Keywords*-Language-based Security, Capabilities, Operational Semantics, JavaScript.**

I. Introduction

An increasing number of current web sites serve active content (applications) from third parties, allowing rich interaction between the embedding page and the embedded applications. Some prominent examples are OpenSocial [1] platforms, iGoogle [2], Facebook [3], and the Yahoo! Application Platform [4], which allow third parties (users of the site) to build JavaScript applications that will be served to other users. While past research [5], [6] has analyzed and improved commercially available methods for providing JavaScript isolation, one fundamental problem remains. Specifically, as illustrated by previously unpublished limitations of Yahoo! ADsafe [7] and Facebook FBJS [8] isolation given in this paper, isolating key elements of a hosting page from applications does not isolate applications from each other. We therefore seek systematic, provably sound methods for designing and verifying isolation *between* untrusted web applications, when displayed as part of a hosting page that interacts with these applications.

Inspired by Google Caja [9], a more complex and sophisticated method than Yahoo! ADsafe or Facebook FBJS, we investigate capability-based methods for providing language-based isolation. Capability-based protection is a widely known method for operating-system-level protection, deployed in such systems as the Cambridge CAP Computer, the Hydra System, StarOS, IBM System/38, the Intel iAPX423, the Amoeba operating system, and others (see [10], [11]). The main idea is that code possessing a capability, such as an unforgeable reference to a file or system object, is allowed to access the resource by virtue of possessing the capability. If a system is *capability safe,* and a process possesses only the capabilities that it is explicitly given, then isolation between two untrusted processes may be achieved granting them non-overlapping capabilities.

An attractive adaptation to programming language contexts is the *object-capability model* [12], [13], which replaces the traditional subject-object dichotomy with programming language objects that are both subjects that initiate access and objects (targets) of regulated actions. Some languages that have been previously designed as object-capability languages are E [14], Joe-E [15], Emily [16], and W7 [17]. Each of these is a restriction or specialized use of a larger programming language, intended to provide capability safety by eliminating language constructs that could leak authority. Specifically, E and Joe-E are restrictions of Java, Emily is a restricted form of OCaml, and W7 is based on Scheme. However, no previous study of these languages related the object-capability model to the semantics of programs in a way that would support rigorous proofs of properties of actual code.

Our original goal was to develop suitable foundations and study whether Cajita [18] or other Caja-based subsets of JavaScript are capability safe and therefore support provably sound isolation between JavaScript applications. In the process of formalizing concepts needed to characterize reachability and isolation using operational semantics of programming languages, we identified a subset of the object-capability goals that we call *authority safety.* Authority safety is sufficient to provide isolation, and also may be achieved by languages that do not support the full object-capability model.

Two access principles articulated in the object-capability literature (e.g., [13], [19], [20]) are "only connectivity begets connectivity" and "no authority amplification." Intuitively, the first condition means that all access must derive from previous access, or, if two sections of code have disjoint or "disconnected" authority, they cannot interfere with each other. The second property restricts the change in authority that may occur when a section of code executes and potentially transfers authority to another: authority is limited to

1081-6011/10 $26.00 © 2010 IEEE
DOI 10.1109/SP.2010.16

initial authority, authority received through interaction, and new authority over newly created resources. Since these two principles are sufficient to bound the authority of executing code, we formalize these two principles using operational semantics of programs and say that a language is authority safe if these two properties are guaranteed. We also give a general proof, based on properties of operational semantics that are satisfied by our example languages and many others, that isolation may be achieved in authority safe languages. As an application of this general theory, we identify a subset of JavaScript and show that it is authority safe, and therefore adequate to support isolation between untrusted applications. This language *Js* is similar to a language presented in [6], but with specific heap initialization code and with native functions and native objects restricted to read-only access.

Although authority safety provides insight into to the isolation problem, the object-capability model provides a structured approach to managing authority that can be used as the basis of programming language design. In other words, authority safety provides safety conditions supporting isolation, but not a programming model or enforcement method. We therefore formalize a form of object-capability model, focussing on reachability properties, in the context of operational semantics of imperative languages and compare capabilities with authority principles. Our main results about this form of capability model are a relatively straightforward theorem that capability safety implies authority safety, and capability safety for a Caja-based subset of JavaScript. While Caja uses a surface language Valija that is translated to a capability safe subset, Cajita, of JavaScript, we focus on the Cajita subset and leave analysis of translations into Cajita for future work

The remainder of this paper is structured as follows. In Section 2, we discuss isolation failures in ADsafe and FBJS. Section 3 presents the semantic framework used to define authority and capability properties precisely, and defines isolation properties. Authority safety is defined and its relation to the mashup isolation problem is formalized in Section 4. Section 5 provides a formal definition of object-capabilities, focussing on reachability, in the context of an operational semantics for imperative code. Sections 6 and 7 contain the main language examples and safety proofs: our authority safe subset, *Js*, and a Caja-based capability safe subset of JavaScript. We discuss related work in Section 8 and give Concluding remarks in Section 9. Technical details omitted due to space limitations can be found in [21].

II. Isolation Failures in ADsafe and Facebook

The need for formal analysis of authority- and capability safety is motivated by some previously unknown vulnerabilities, described in this Section. These vulnerabilities, discovered while analyzing the subsets of JavaScript enforced by Yahoo! ADsafe and Facebook FBJS from an *authority safety* perspective, would be prevented by authority (and/or capability) tracking as developed in later sections of the paper.

A. ADsafe

The ADsafe subset [7] is designed to protect hosting pages from embedded advertising code by limiting access to the DOM through a combination of static analysis and syntactic restrictions. The advertising JavaScript code must satisfy severe syntactical restrictions: for example it cannot include the keyword this, which could otherwise be used to access the global object and gain direct access to the DOM. Restricted code instead must use an ADSAFE object that serves as a library to mediate access to the DOM and other page services.

Examining ADsafe from the point of view of a capability-based approach, we found three problematic cases that let untrusted ad code violate the ADsafe sandbox. We disclosed these attack cases to Doug Crockford, who promptly fixed the ADsafe platform.

The first problem we found is that a library function exposed to the untrusted code was defined to return its this parameter when invoked without arguments. The following code is valid ADsafe code that escapes the sandbox, accesses the global object and raises an alert “Hacked!”.

```
var a = dom.tag("div").ephemeral;
var asd = a().alert("Hacked!");
```

A sound definition of authority for ADsafe code would immediately flag that the authority to read ephemeral and to call it implies the authority to read the global object.

The second problem we found is related to an attack on Facebook reported in [6], and exploits the JavaScript implicit type conversion mechanism. The ADsafe dom.tag function, which ad code can use to create any non-dangerous DOM element, can be forced to create a script element, which amounts to inject arbitrary code in the page.

```
var o = {toString:function(){o.toString =
                                  function(){return "script"};
                              return "div"}};
dom.append(dom.tag(o).append(dom.text("alert('Hacked!')")));
```

The attacks works because internally dom.tag had a structure similar to if (o!="script"){return document.createElement(o);}, and our object o is able to “lie” the first time it is converted to a string, claiming to be a div element. The call to document.createElement(o); has the authority to create any tag element that o can evaluate to. A capability safe alternative would be to limit such authority by construction, for example using an idiom like

```
var tag= {"div":"div","script":"text"}[o];
return document.createElement(tag);
```

Finally, we found an example where a library function exposed to the user returned an object that leaked the

authority to read directly DOM objects. From a DOM object, untrusted code can get access to the enclosing document by accessing the ownerDocument property.

```
var a = dom.text(["hacked"]);
a[0].ownerDocument.location="http://attacker.com";
```

These examples are do not indicate a fundamental flaw in the ADsafe design, but do suggest the need for a systematic formal analysis of isolation properties and the means to achieve them.

B. FBJS

Facebook [3] is a well-known social networking web site that allows users to store and share private and public information. A common way for Facebook users to interact is through Facebook applications, which are retrieved from the application publisher's server, filtered by Facebook servers and embedded in the user page. The scripts used by such applications must be written in a subset of JavaScript called FBJS [8] that restricts them from accessing arbitrary parts of the DOM tree of the larger Facebook page.

While the details of the FBJS restrictions have been studied in detail elsewhere [22], we shall here simply note that a crucial part of these restrictions consist of preventing untrusted code from accessing directly the global object, and therefore all its properties that reference native JavaScript objects (such as Object, Function, Object.prototype, etc.). In fact, tampering with those objects amounts to tampering with the execution environment of the whole browser window, including the Facebook libraries and other user applications.

Authority leaks. While trying to define the authority provided to FBJS expressions, we realized that the actual FBJS implementation does not seem to take into account the authority conveyed by the prototype-based inheritance mechanism of JavaScript. The simple expression

```
var Obj = {};
var ObjProtToString = Obj.toString;
```

illustrates this problem, by saving in the variable ObjProtToString the heap address of the toString function found in the native Object.prototype. A FBJS application can now read or write properties of that native function at will.

This example shows how, by traversing the prototype chain, some FBJS code with no capabilities to modify native objects can gain this capability without being granted the authority by another principal. We illustrate the ramification of this kind of capability leakage with two examples.

Communication channels between FBJS applications. Two Facebook applications can easily (and unsafely) exchange data or code with each other, bypassing the FBJS runtime altogether. The sending application can execute the code

```
({}).toString.channel = "message";
```

```
<a href="#" onclick="break()">Attack FBJS!</a>
<script>
function break(){
  var f = function(){};
  f.bind.apply =
      (function(old){return function(x,y){
                          var getWindow = y[1].setReplay;
                          getWindow(0).alert("Hacked!");
                          return old(x,y)}
        })(f.bind.apply)}
</script>
```

Figure 1. Exploit code.

The receiving one can read the data back from the channel

```
var a67890_message = ({}).toString.channel;
```

Note that channel is a newly created public property of the shared function/object Object.prototype.toString. Essentially, this examples shows a way for untrusted code to gain write capabilities on the properties of the properties of native prototype objects.

While this communication mechanism could be considered useful by application writers, it is not a documented FBJS feature, and we do not believe was intended by the FBJS designers. In the example above the two applications are willing to communicate with each other. It is possible to construct other examples where an application maliciously alters the behavior of another without the latter being aware. The next example supersedes that case, showing that this leak of authority can be used to compromise the whole FBJS runtime environment.

Compromising the Facebook sandbox. A programming pattern common in both FBJS applications and runtime libraries consists of calling functions through the standard call and apply methods rather than directly (i.e. using f.call(o,v) rather than o.f(v)). Another common idiom is to use a bind method to curry the arguments of a function. What if a malicious application redefines for example the apply method of the bind function of Function.prototype? Then, the attacker can automatically hijack all the function invocations arising from expressions of the form f.bind.apply(e), and have full access to e. This is indeed the basis of our attack that can break the FBJS sandbox.

We found that when the Facebook page receives an AJAX message, it invokes a function fitting that pattern, and it passes as argument an object from which we can steal a direct reference to the window object. The details of how that can be done are complicated and not necessary for the current discussion. The important point is that untrusted code is not supposed to have the capabilities to modify such internal functions, and should not be able to gain them. In Figure 1 we report the exploit code of the malicious

Facebook application, which raises an alert dialogue with the message "Hacked!". The serious potential implications of Facebook exploits have been discussed in the literature [23], [22]. We described this problem to the Facebook team and proposed a solution based on avoiding the nested call pattern f.bind.apply(e) in FBJS libraries. The problem is currently being fixed.

As we shall see below, a safe approximation of authority for JavaScript terms would make these examples of insecure behaviour very easy to spot.

III. Isolation for Mashups

We consider isolation between components supplied by different principals, in a general language setting. As explained below, we assume that the programming language has a deterministic operational semantics of a given form. For concreteness, and to simplify some of the technical analysis, we consider a basic form of mashup based on sequential composition that appears to provide the basic challenges present whenever there is no preemptive scheduling. In this context, the basic isolation problem, defined more precisely in Section III-B is to ensure that during the execution of a mashup, execution of one component does not access certain security-critical resources of the execution environment or interfere with the execution of any other component.

All the languages considered in the rest of this paper are assumed to satisfy the conditions given in Section III-A. The operational semantics of JavaScript developed in [5], [22], [6] meets all of these conditions.

A. Assumptions

We assume a programming language $\mathfrak{L}$ with a deterministic small-step operational semantics $\mathfrak{S}$ satisfying the conditions below.

States, heaps and terms. We assume that a state $S = (H, t)$ consists of a *heap* H representing program memory and a *term* t representing the program being executed. The heap is assumed to be a mapping from *heap addresses* to values (which could be language specific). Depending on the structure of the heap and the language semantics, a term may be able to write to or read from specific portions of the memory denoted by a heap address. We refer to the smallest such portions of memory as *resources*. We let $\mathbb{R}$ be the set of all possible resources and write $\mathsf{res}(H)$ for the set of resources associated with a given heap H. For most languages, $\mathsf{res}(H)$ would be the set of allocated heap addresses, but our analysis allows any definition of $\mathsf{res}(H)$ as long as remaining conditions are satisfied.

We let $\mathbb{T}_{\mathfrak{L}}$ be the set of all syntactically well-formed terms of the language and let $\mathbb{I}_{\mathfrak{L}}$ be a set of principals that may supply terms to a mashup. The set of terms may include user terms and intermediate terms that are used simply to express the operational semantics and may produced during the evaluation of user terms: $\mathbb{T}_{\mathfrak{L}} := \mathbb{T}^{\mathsf{u}}_{\mathfrak{L}} \uplus \mathbb{T}^{\mathsf{i}}_{\mathfrak{L}}$. The subterm relation $\sqsubseteq$ is the usual partial order on the structure of terms. Given a set of terms, the term contexts are syntactically wellformed terms with one or more place holders $[-]$, usually generated by the grammar of the language.

Operational semantics. For any state S, we let $\mathsf{heap}(S)$ and $\mathsf{term}(S)$ be its heap and term components. We assume that the semantic evaluation rules are of two kinds

- Base Case rules: $\frac{\mathsf{P}(H,t)}{H,t \to K,s}$ - where $\mathsf{P}(H,t)$ is a predicate that must be true for the rule to be enabled.
- Contextual rules: $\frac{H,t \to K,s}{H,C[t] \to K,C[s]}$ - where $C[-]$ is an evaluation context.

Given a state S, the *trace* $\tau(S)$ is the possibly infinite sequence of states obtained evaluating S. We write $S \uparrow$ when $\tau(S)$ is non-terminating and $\mathsf{final}(\tau)$ for the final state if trace τ is finite. We assume that for every final state S, the term $\mathsf{term}(S)$ is always a value (i.e., cannot be evaluated further). We partition the set of all possible values into the set of normal values $\mathbb{V}$ and the set of error values $\mathbb{E}$.

Writing $\mathsf{Wf}(S)$ to indicate that state S is well formed (for example, every location named in $\mathsf{term}(S)$ has a value in $\mathsf{heap}(S)$), we assume that the set of wellformed states is closed under evaluation. In addition, we assume that Wf is compositional on terms (for example, $\mathsf{Wf}(H, t_1; t_2)$ iff $\mathsf{Wf}(H, t_1)$ and $\mathsf{Wf}(H, t_2)$) and depends on a wellformedness predicate for heaps Wf^{h} (in particular, $\mathsf{Wf}(H, t)$ implies $\mathsf{Wf}^{\mathsf{h}}(H)$). We write $\mathbb{H}$ for the set of all wellformed heaps in the language $\mathfrak{L}$. We assume that the language semantics provides an initial heap $H_{\mathfrak{L}} \in \mathbb{H}$, where computation begins.

Sequential composition. We assume that the language supports sequential composition ";" with the semantics described below, which is a generalization of the standard semantics for most imperative languages with exceptions. Given a wellformed heap H_1 and terms t_1 and t_2 such that $\mathsf{Wf}(H_1, t_1; t_2)$ holds, either $H_1, t_1; t_2$ diverges or there must exist an evaluation context C_1 and a heap H_2 such that

- $H_1, t_1; t_2 \to^* H_1, C_1[t_1] \to^* H_2, C_1[v] \to^* H_2, t$;
- if $v \in \mathbb{V}$ then $t = C_2[t_2]$ for some C_2;
- if $v \in \mathbb{E}$ then $t = v$.

Intuitively, the semantics may provide some bookkeeping context $C_1[-]$ to evaluate t_1 to a value v, and then it can either continue with the evaluation of t_2 (possibly in another bookkeeping context $C_2[-]$), or report an error.

B. The Isolation Problem

Intuitively, the isolation problem is to ensure that in any mashup, execution of one component does not access certain security-critical resources of the execution environment or interfere with the execution of any other component. Since this depends on the definition of mashup, we begin with mashups. For simplicity, a mashup is defined by sequential composition of mashup components, as follows.

Definition 1 (Basic Mashup): A basic mashup

$$\mathsf{Mashup}((t_1, id_1), \ldots, (t_n, id_n))$$

is an ordered list of pairs of terms and principals, with $t_1, \ldots, t_n \in \mathbb{T}_{\mathfrak{L}}$ *and* $id_1, \ldots, id_n \in \mathbb{I}_{\mathfrak{L}}$. *The* program $\mathsf{prog}(m)$ *is the sequential composition* $t_1; \ldots; t_n$ *of the terms.*

Given a mashup $m = \mathsf{Mashup}((t_1, id_1), \ldots, (t_n, id_n))$ and a heap H with $\mathsf{Wf}(H, \mathsf{prog}(m))$, the sequence $\mathsf{states}(H, m, id_i)$ records the states in $\tau(H, \mathsf{prog}(m))$ that arise from executing term t_i provided by principal id_i. If the evaluation of a component t_k leads to an error state or diverges, then $\mathsf{states}(H, m, id_i)$ is empty for the principals with index $i > k$.

The sequence $\mathsf{states}(H, m, id_i)$ may be defined more precisely as follows. Let k be the least natural number with $\mathsf{term}(\mathsf{final}(\tau(H, t_1; \ldots; t_i)) \notin \mathbb{V}$, meaning that $t_1; \ldots; t_i$ does not terminate normally.

- For $i < k$, $\mathsf{states}(H, m, id_i)$ is the subtrace $H_i, C_i[t_i] \rightarrow^* H_{i+1}, C_i[v_i]$ of the computation $H_m, \mathsf{prog}(m) \rightarrow^* H_m, C_1[t_1] \rightarrow^* H_2, C_1[v_1] \rightarrow^* H_2, C_2[t_2] \rightarrow^* H_3, C_2[v_2] \rightarrow^* \ldots \rightarrow^* H_k, C_k[t_k]$.
- Let $\mathsf{states}(H, m, id_k)$ be the trace $\tau(H_k, C_k[t_k])$.
- For $i > k$, let $\mathsf{states}(H, m, id_i) = \emptyset$.

We will define isolation using the set of heap-affecting actions performed during a single step of evaluation. Let $\mathbb{D} = \{\mathsf{r}, \mathsf{w}\}$ have two tokens, indicating read and write permissions, and let $\mathbb{A} := \mathbb{R} \times \mathbb{D}$ be the set of all read and write permissions on resources. For any heap H, the set of heap-affecting actions is $\mathsf{act}(H) := \mathsf{res}(H) \times \mathbb{D}$, where (p, r) (or (p, w)) denotes reading (or writing) the value at position p. It is straightforward to partition $\mathsf{act}(H)$ into read and write actions $\mathsf{act}^{\mathsf{r}}(H)$ and $\mathsf{act}^{\mathsf{w}}(H)$. We consider creation of a new empty resource as a read action, and the initialization of such resource as a write action.

Definition 2 (Can Influence): A write action $a_1 = (p, \mathsf{w})$ can influence *read action* $a_2 = (p, \mathsf{r})$ *on the same resource* $p \in \mathsf{res}(H)$, *written* $a_1 \triangleright a_2$. *Set* $\mathcal{A}_1$ *of heap actions* can influence *set* $\mathcal{A}_2$, *written* $\mathcal{A}_1 \triangleright \mathcal{A}_2$ *if* $\exists a_1 \in \mathcal{A}_1, a_2 \in \mathcal{A}_2$ *such that* $a_1 \triangleright a_2$.

Definition 3 (Access): For any well-formed state S, $\mathsf{acc}(S)$ *is the set of all actions* (p, r) *or* (p, w) *such that resource* p *is read or written, respectively, during the single evaluation step from state* S. *The* acc *function is naturally extended to traces and sets of states.*

The set of all actions performed during the complete evaluation of S is therefore $\mathsf{acc}(\tau(S))$. Given a heap H_m, a mashup m and a principal id_k, the actions performed by the component id_k can be expressed $\mathsf{acc}(\mathsf{states}(H_m, m, id_k))$.

Informally, isolation of mashups includes two properties: (i) the actions performed by the individual components are mutually non-influencing; (ii) the set of actions performed by each component do not include certain forbidden heap actions. The forbidden actions are generally actions accessing security critical resources specified by the hosting environment for the mashup. In our analysis, assume a set $\mathcal{A}_{\varnothing}$ of forbidden heap actions.

Definition 4 (Isolation Property): Given a heap H_m, *a mashup* $m = \mathsf{Mashup}((t_1, id_1), \ldots, (t_n, id_n))$ *such that* $\mathsf{Wf}(H_m, \mathsf{prog}(m))$, *and a set* $\mathcal{A}_{\varnothing}$ *of forbidden heap actions, we have* $\mathsf{Isolation}(\mathcal{A}_{\varnothing}, H_m, m)$ *if*

1) $\forall i, j : i < j \Rightarrow$ $\mathsf{acc}(\mathsf{states}(H_m, m, id_i)) \not\triangleright \mathsf{acc}(\mathsf{states}(H_m, m, id_j))$;
2) $\forall i : \mathsf{acc}(\mathsf{states}(H_m, m, id_i)) \cap \mathcal{A}_{\varnothing} = \emptyset$.

Isolation approach. Since arbitrary mashups may not satisfy the isolation property, it is useful to be able to enforce mashup isolation by the following steps, also followed by FBJS and ADsafe: (i) Perform a source-to-source translation of each component t_i thereby confining its behavior to a restricted set of resources, and (ii) Set up an appropriate initial environment so that the resources accessible to the various components do not *leak* any access to the forbidden resources and also do not act as communication channels between the various components.

Writing $\mathsf{enf}^i : \mathbb{T}_{\mathfrak{L}} \rightarrow \mathbb{T}_{\mathfrak{L}}$ for a source-to-source translation applied to the component supplied by principal id_i, we can define the isolation problem as follows:

Problem 1 (Isolation Problem): Given a set of forbidden actions $\mathcal{A}_{\varnothing}$, *define an initial heap* H_m *and enforcement functions* $\mathsf{enf}^1, \ldots, \mathsf{enf}^n$, *such that for all terms* $t_1, \ldots, t_n \in \mathbb{T}_{\mathfrak{L}}$ *provided by principals* $id_i, \ldots, id_n$, *if* $\mathsf{Wf}(H_m, \mathsf{prog}(m))$ *holds, then* $\mathsf{Mashup}((\mathsf{enf}^1(t_1), id_1), \ldots, (\mathsf{enf}^n(t_n), id_n))$ *satisfies the isolation property* $\mathsf{Isolation}(\mathcal{A}_{\varnothing}, H_m, m)$.

IV. Authority Safety

In this Section, we identify properties based on object-capability systems that are sufficient to ensure isolation, as characterized in Section III. In programming languages, the *subjects* that access resources are program terms, and resources, as described in the previous Section, are the smallest granularity readable/writable positions on the program heap. We allow the *authority of a subject* to be any upper bound on the set of actions that a subject can perform on all resources. An important part of the isolation problem is that the authority of a subject may change due to actions performed by other subjects, if they share resources. For example, a term with access to a linked list will gain authority if another term adds to the linked list. Although there is no prior precise semantic definition of *capability safe languages*, we draw inspiration from a useful body of work on object-capability models and their properties.

A fundamental question we want to answer is: *What property of capability safe languages makes them favorable for enforcing isolation between mutually mistrusting subjects?*

We propose AuthoritySafe as a candidate answer to this question.

Authority safety. We say that a programming language is AuthoritySafe if it satisfies the following properties associated with the object-capabilities model:

1) **Only connectivity begets connectivity:** A subject can influence the authority of only those subjects whose authority influences its own authority.
2) **No authority amplification:** The change in authority of a subject due to actions performed by another subject is bounded by the authority of the acting subject.

Although derived from the object-capability model, these properties are actually independent of the specific details of the object-capability model, and can be independently evaluated for a general sequential programming language. In Section V, we define a form of capability safety and show that capability safety implies AuthoritySafe.

After defining AuthoritySafe more precisely below, we prove that for any two subjects t_1 and t_2 provided by principals id_1 and id_2, if the initial authorities of t_1 and t_2 are non-influencing, then the evaluation of term t_1 cannot influence the evaluation of term t_2 in the mashup $\mathsf{Mashup}((t_1, id_1), (t_2, id_2))$. This result allows us to show that isolation may be achieved by performing an initial source to source translation in order to ensure mutually non-influencing executions of all the components of the mashup.

A. Formalizing AuthoritySafe

We define authority safety using the notion of "authority of a term". Authority is informally a function of a heap H and a term t which expresses an over-approximation of the heap actions that are performed in the trace of the state H, t.

Definition 5 (Valid Authority map): A map $\mathsf{auth} : \mathbb{H} \times \mathbb{T}_{\mathfrak{L}} \rightarrow \mathbb{A}$ *is said to be a valid authority map if, for all states* H, t *and* K, s*, whenever* $H, t \rightarrow K, s$*, the following holds:*

$$\begin{array}{rcl} \mathsf{acc}(H,t) & \subseteq & \mathsf{auth}(H,t) \cup (\mathsf{act}(K) \setminus \mathsf{act}(H)); \\ \mathsf{auth}(K,s) & \subseteq & \mathsf{auth}(H,t) \cup (\mathsf{act}(K) \setminus \mathsf{act}(H)). \end{array}$$

We now define AuthoritySafe, assuming a valid authority map.

*Definition 6 (*AuthoritySafe *property): A language* $\mathfrak{L}$ *is* authority safe for *authority map* auth *if, for all well-formed states* H, t *and* K, v *with* $H, t \rightarrow^* K, v \not\rightarrow$ *and* $v \in \mathbb{V}$*, and for all* $u \in \mathbb{T}_{\mathfrak{L}}$ *such that* $\mathsf{Wf}(H, u)$ *and* $\mathsf{Wf}(K, u)$*, the following holds:*

1) ***Connectivity begets connectivity:***

$$\mathsf{acc}(\tau(H,t)) \not\triangleright \mathsf{auth}(H,u) \Rightarrow \mathsf{auth}(H,u) = \mathsf{auth}(K,u)$$

2) ***No authority amplification:***

$$\mathsf{acc}(\tau(H,t)) \triangleright \mathsf{auth}(H,u) \quad \Rightarrow$$
$$\mathsf{auth}(K,u) \subseteq \mathsf{auth}(H,u) \cup \mathsf{auth}(H,t) \cup (\mathsf{act}(K) \setminus \mathsf{act}(H)).$$

B. Solving the Mashup Isolation Problem

We now show that the authority safety of a language helps solving the mashup isolation problem. First of all we define the authority isolation property, which formalizes the idea that some terms do not have sufficient authority to influence each other, or access forbidden properties.

Definition 7 (Authority Isolation): Consider an authority safe language $\mathfrak{L}$ *with authority map* auth*. The terms* $t_1, \ldots, t_n \in \mathbb{T}_{\mathfrak{L}}$ *enjoy* authority isolation *with respect to the wellformed heap* H *and the set of forbidden actions* $\mathcal{A}_\varnothing$*, denoted by* $\mathsf{AuthorityIsolation}(\mathcal{A}_\varnothing, H, t_1, \ldots, t_n)$*, if the following properties hold:*

1) $\forall i, j : i < j \Rightarrow \mathsf{auth}(H, t_i) \not\triangleright \mathsf{auth}(H, t_j)$.
2) $\forall i : \mathsf{auth}(H, t_i) \cap \mathcal{A}_\varnothing = \emptyset$.

We now show that AuthorityIsolation implies Isolation for a mashup.

Theorem 1: Consider an authority safe language $\mathfrak{L}$ *with authority map* auth*. Given a set of forbidden actions* $\mathcal{A}_\varnothing$*, a mashup* $m = \mathsf{Mashup}((t_1, id_1), \ldots, (t_n, id_n))$*, and a heap* H_m *such that* $\mathsf{Wf}(H_m, \mathsf{prog}(m))$ *holds,*

$$\mathsf{AuthorityIsolation}(\mathcal{A}_\varnothing, H_m, t_1, \ldots, t_n) \quad \Rightarrow \quad \mathsf{Isolation}(\mathcal{A}_\varnothing, H_m, m).$$

Proof Sketch: Consider the valid authority map auth for which AuthoritySafe holds. In order to prove the theorem, we define the sequence of heaps $H_1, \ldots, H_k$ ($k \leq n$) such that $H_1 = H_m$ and $\forall i < k : \mathsf{term}(\mathsf{final}(\tau(H_i, t_i))) \in \mathbb{V}$ and $H_{i+1} = \mathsf{heap}(\mathsf{final}(\tau(H_i, t_i)))$, and $\mathsf{term}(\mathsf{final}(\tau(H_k, t_k))) \in \mathbb{E}$ or $H_k, t_k \uparrow$.
Since $\mathsf{Wf}(H_m, t_1; \ldots; t_n)$ holds, and wellformedness is closed under evaluation, for all i, j we have

$$i \leq j \Rightarrow \mathsf{Wf}(H_i, t_j). \tag{1}$$

From the definition of states and the definition of contexts rules, for all i, k we have

$$i \leq k \Rightarrow \mathsf{acc}(\mathsf{states}(H_m, m, id_i)) = \mathsf{acc}(\tau(H_i, t_i)); \tag{2}$$
$$i > k \Rightarrow \mathsf{acc}(\mathsf{states}(H_m, m, id_i)) = \emptyset. \tag{3}$$

From AuthoritySafe(auth), we have $\mathsf{acc}(\tau(H_i, t_i)) \subseteq \mathsf{auth}(H_i, t_i)$. Therefore, in order to prove $\mathsf{Isolation}(\mathcal{A}_\varnothing, H, m)$, it is sufficient to show, for all i, j

$$1 \leq i < j \leq k \quad \Rightarrow \quad \mathsf{auth}(H_i, t_i) \not\triangleright \mathsf{auth}(H_j, t_j); \tag{4}$$
$$1 \leq i \leq k \quad \Rightarrow \quad \mathsf{auth}(H_i, t_i) \cap \mathcal{A}_\varnothing = \emptyset. \tag{5}$$

From $\mathsf{AuthorityIsolation}(\mathcal{A}_\varnothing, H, t_1, \ldots, t_n)$, we have

$$1 \le i < j \le k \quad \Rightarrow \quad \mathsf{auth}(H_m, t_i) \not\vdash \mathsf{auth}(H_m, t_j); \quad (6)$$

$$1 \le i \le k \quad \Rightarrow \quad \mathsf{auth}(H_m, t_i) \cap \mathcal{A}_\varnothing = \emptyset. \quad (7)$$

Properties 4 and 5 will follow from Properties 6 and 7 if we show that $\mathsf{auth}(H_i, t_i) = \mathsf{auth}(H_m, t_i)$ for all $i \in 1..k$. This follows by proving the stronger property

$$\forall i, j \in 1..k : i \le j \Rightarrow \mathsf{auth}(H_i, t_j) = \mathsf{auth}(H_m, t_j)$$

by induction on i. □

Using Theorem 1, the isolation problem can be reduced to defining an initial heap H_m and enforcement functions $\mathsf{enf}^1, \ldots, \mathsf{enf}^n$, such that, for all terms $t_1, \ldots, t_n \in \mathbb{T}_{\mathfrak{L}}$ $\mathsf{AuthorityIsolation}(\mathcal{A}_\varnothing, H_m, \mathsf{enf}^1(t_1), \ldots, \mathsf{enf}^n(t_n))$ holds.

V. Object Capability Systems

Building on the formal definitions of Section III, in this Section we present a definition of capability safety in the context of a sequential programming language. We focus on reachability properties of the object-capability model, leaving further formalization and analysis of the object-capability model to later work.

Before proposing a rigorous definition of capability safety we surveyed various existing capability safe languages. A general capability system for a programming language, informally, refers to the notions of *resources*, *subjects*, *capabilities* and *authority associated with capabilities*. We have already encountered resources and subjects in Section IV. Capabilities are unspoofable and unforgeable entities which, when possessed by a subject, grant it the ability to designate and access one or more resources in a specific way. The authority associated with a capability is an over-approximation of all the actions that can be performed by *exercising* the capability in all possible ways. In a capability system, the authority associated with a subject is essentially the cumulative authority associated with all the capabilities possessed by the subject. Additionally, the capability system imposes specific restrictions on how the actions performed by a subject can affect the resources and the authority associated with every capability present in the system.

Object-capability systems are a very important special case of capability systems. These systems contain a central notion of *object*, which is an entity that encapsulates resources together with the code for accessing those resources. Objects essentially form the set of capabilities in such systems. Each object may possess references to other objects and therefore a reference graph of objects can be defined. The authority associated with an object is determined by the behavior of the code stored in the object. An upper-bound on this authority is given by the union of the set of resources contained in all objects that are reachable from a given object in the reference graph. This notion is also known as the *topology-only bound on authority*.

Formal definitions. Our formal definition of capability system is close in spirit to the object-capability model. Capabilities are entities which designate and carry read or write (or both) permissions to access a specific resource. A capability system is composed of a set of capabilities ($\mathbb{C}$) together with functions to obtain the resources designated by each capability (desg), the privileges associated to each capability (priv), the set of capabilities associated with a term in the language (tCap), the set of all capabilities which designate resources present on a particular heap (hCap) and the authority associated with a capability (cAuth) in a specific heap.

Definition 8 (Capability System): A capability system *is a tuple consisting of:*

- *a set* $\mathbb{C}$*;*
- *a function* $\mathsf{desg} : \mathbb{C} \to \mathbb{R}$*;*
- *a function* $\mathsf{priv} : \mathbb{C} \to 2^{\mathbb{D}}$*;*
- *a function* $\mathsf{tCap} : \mathbb{T}_{\mathfrak{L}} \to 2^{\mathbb{C}}$*;*
- *a function* $\mathsf{hCap} : \mathbb{H} \to 2^{\mathbb{C}}$*;*
- *a function* $\mathsf{cAuth} : \mathbb{H} \times \mathbb{C} \to 2^{\mathbb{A}}$.

In order to denote a valid capability system, the tuple must satisfy the following conditions:

1) ***Basic conditions:*** *For all* $s, t \in \mathbb{T}_{\mathfrak{L}}$*,* $H \in \mathbb{H}$ *and* $c \in \mathbb{C}$
   a) $s \sqsubseteq t \;\Rightarrow\; \mathsf{tCap}(s) \subseteq \mathsf{tCap}(t)$
   b) $\mathsf{desg}(c) \in \mathsf{res}(H) \;\Leftrightarrow\; c \in \mathsf{hCap}(H)$.
   c) $(c \notin \mathsf{hCap}(H) \vee \mathsf{priv}(c) = \emptyset) \;\Rightarrow\; \mathsf{cAuth}(H, c) = \emptyset$
   d) $(c \in \mathsf{hCap}(H) \wedge \mathsf{priv}(c) \neq \emptyset) \;\Rightarrow\; \{\mathsf{desg}(c)\} \times \mathsf{priv}(c) \subseteq \mathsf{cAuth}(H, c) \subseteq \mathsf{act}(H)$
   e) $\mathsf{Wf}(H, t) \;\Rightarrow\; \mathsf{tCap}(t) \subseteq \mathsf{hCap}(H)$.
2) ***Topology-only bound for*** cAuth***:*** *For all* $H \in \mathbb{H}$ *and* $c \in \mathbb{C}$*, with* $\mathcal{C} = \mathsf{tCap}(H(\mathsf{desg}(c)))$
   a) $\mathsf{r} \in \mathsf{priv}(c) \;\Rightarrow\; \mathsf{cAuth}(H, c) \subseteq \{\mathsf{desg}(c)\} \times \mathsf{priv}(c) \;\cup\; \bigcup_{c' \in \mathcal{C}} \mathsf{cAuth}(H, c')$
   b) $\mathsf{r} \notin \mathsf{priv}(c) \;\Rightarrow\; \mathsf{cAuth}(H, c) \subseteq \{\mathsf{desg}(c)\} \times \mathsf{priv}(c)$.

We now define the conditions under which a capability system is safe. These conditions are essentially restrictions on how a subject can affect the resources and the authority of all capabilities.

Definition 9 (Capability Safety): A language $\mathfrak{L}$ *is* capability safe *with respect to the capability system* $(\mathbb{C}, \mathsf{desg}, \mathsf{priv}, \mathsf{tCap}, \mathsf{cAuth}, \mathsf{hCap})$ *if for all wellformed states* H, t *the following conditions hold:*

1) *The map* $\mathsf{auth}(H, t) = \bigcup_{c \in \mathsf{tCap}(t)} \mathsf{cAuth}(H, c)$ *is a valid authority map for the language.*
2) *For all wellformed states* K, v *such that* $H, t \to^* K, v \not\to \;\wedge\; v \in \mathbb{V}$*, and for all capabilities* $c \in \mathsf{hCap}(K)$

$$\text{a)}\ \ \mathsf{acc}(H,t) \not\triangleright \mathsf{cAuth}(H,c) \quad \Rightarrow$$
$$\mathsf{cAuth}(K,c) = \mathsf{cAuth}(H,c) \ \cup\ (\{\mathsf{desg}(c)\} \times \mathsf{priv}(c));$$
$$\text{b)}\ \ \mathsf{acc}(H,t) \triangleright \mathsf{cAuth}(H,c) \quad \Rightarrow$$
$$\mathsf{cAuth}(K,c) \subseteq \mathsf{cAuth}(H,c) \cup (\{\mathsf{desg}(c)\} \times \mathsf{priv}(c)) \cup (\textstyle\bigcup_{c' \in \mathsf{tCap}(t)} \mathsf{cAuth}(H,c')) \cup (\mathsf{act}(K) \setminus \mathsf{act}(H)).$$

Consider any capability safe language $\mathfrak{L}$ with capability system $(\mathbb{C}, \mathsf{desg}, \mathsf{priv}, \mathsf{tCap}, \mathsf{cAuth})$. For any wellformed heap H, we define the capability graph $\mathcal{G}^{\mathsf{C}}(H)$ as a graph with nodes as capabilities from the set $\mathsf{hCap}(H)$ and an edge between capabilities c_1 and c_2 iff $r \in \mathsf{priv}(c_1) \wedge c_2 \in \mathsf{tCap}(H(\mathsf{desg}(c_1)))$. We denote by $c_1 \rightsquigarrow_H c_2$ a path from node c_1 to node c_2 in the graph $\mathcal{G}^{\mathsf{C}}(H)$.

We now give a proposition which precisely states the conditions under which an edge can be added to the capability graph as the heap changes from H to K due to the evaluation of some term t.

Proposition 1: Consider a capability safe language $\mathfrak{L}$ with a capability system $(\mathbb{C}, \mathsf{desg}, \mathsf{priv}, \mathsf{tCap}, \mathsf{cAuth})$. Suppose there exist wellformed states H, t and K, v such that $H, t \rightarrow^* K, v \not\rightarrow \ \wedge\ v \in \mathbb{V}$. One of the following must be true for any two capabilities $c_1, c_2 \in \mathsf{hCap}(K)$ for which there is an edge $c_1 \rightarrow c_2$ in $\mathcal{G}^{\mathsf{C}}(K)$:

1) ***Connectivity by initial conditions:*** *The edge $c_1 \rightarrow c_2$ is present in $\mathcal{G}^{\mathsf{C}}(H)$.*
2) ***Connectivity by introduction:*** *$c_1, c_2 \in \mathsf{hCap}(H)$ and $\exists c_1', c_2' \in \mathsf{tCap}(t) : c_1' \rightsquigarrow_H c_1 \wedge c_2' \rightsquigarrow_H c_2$.*
3) ***Connectivity by parenthood:*** *$c_1 \in \mathsf{hCap}(H) \wedge c_2 \notin \mathsf{hCap}(H)$ and $\exists c_1' \in \mathsf{tCap}(t) : c_1' \rightsquigarrow_H c_1$.*
4) ***Connectivity by endowment:*** *$c_1 \notin \mathsf{hCap}(H) \wedge c_2 \in \mathsf{hCap}(H)$ and $\exists c_2' \in \mathsf{tCap}(t) : c_2' \rightsquigarrow_H c_2$.*

We are now ready to present the main result of this Section, that capability safety implies authority safety.

Theorem 2: A capability safe language $\mathfrak{L}$ with capability system $(\mathbb{C}, \mathsf{desg}, \mathsf{priv}, \mathsf{tCap}, \mathsf{cAuth})$ is authority safe with respect to the authority map $\bigcup_{c' \in \mathsf{tCap}(t)} \mathsf{cAuth}(H, c')$.

Proof Sketch: Since the language $\mathfrak{L}$ is capability safe for the system $(\mathbb{C}, \mathsf{desg}, \mathsf{priv}, \mathsf{tCap}, \mathsf{cAuth})$, we know that $\mathsf{auth}(H,t) := \bigcup_{c \in \mathsf{tCap}(t)} \mathsf{cAuth}(H,c)$ is a valid authority map. We now show that AuthoritySafe holds for auth: namely (i) that *only connectivity begets connectivity* and (ii) that there is *no authority amplification*, according to Definition 6.

Consider any states H, t and K, v such that $H, t \rightarrow^* K, v \not\rightarrow$ and $v \in \mathbb{V}$. Consider also any term $u \in \mathbb{T}_{\mathfrak{L}}$ such that $\mathsf{Wf}(H,u)$ and $\mathsf{Wf}(K,u)$. Below, we prove (i). The proof of (ii) is similar.

Suppose $\mathsf{acc}(\tau(H,t)) \not\triangleright \mathsf{auth}(H,u)$. We need to show that $\mathsf{auth}(H,u) = \mathsf{auth}(K,u)$, that is implied by

$$\forall c \in \mathsf{tCap}(u) \quad : \quad \mathsf{cAuth}(H,c) = \mathsf{cAuth}(K,c). \qquad (8)$$

Since $\mathsf{Wf}(H,u)$ and $\mathsf{Wf}(K,u)$, for any $c \in \mathsf{tCap}(u)$ we have $c \in \mathsf{hCap}(H)$ and $c \in \mathsf{hCap}(K)$. Since $\mathsf{acc}(\tau(H,t)) \not\triangleright \mathsf{auth}(H,u)$, we have $\forall c \in \mathsf{tCap}(u) : \mathsf{acc}(\tau(H,t)) \not\triangleright \mathsf{cAuth}(H,c)$. Condition 8 now follows easily from Condition (2a) of Definition 9 and Condition (1d) of Definition 8, and we conclude. □

VI. *Js* IS AUTHORITY SAFE

In this Section we describe the language *Js* which is a subset of ECMAScript 3 (E_3 in brief), the ECMA-262-compliant version of JavaScript [5]. E_3 was the most recent version of JavaScript when this paper was written, and it was current when existing browsers implementations were developed. The latest version of the standard is ECMAScript 5, which we will consider in future work. We define *Js* as a minor variant of the language J_{sub} of [6] that satisfies the assumptions of Section III-A.

A. Operational Semantics of E_3

Given the space constraints, we only describe (informally) the semantics of some of the unusual and interesting constructs of E_3 which motivate the definition of *Js*.

Heaps and terms. We denote by $\mathbb{T}^{\mathsf{u}}_{E_3}$ the set of all terms derivable from the syntax of E_3. Besides these terms, the semantics introduces some internal terms, objects and properties useful to express the reduction semantics of the language. We denote the set of internal terms by $\mathbb{T}^{\mathsf{i}}_{E_3}$ and define $\mathbb{T}_{E_3} = \mathbb{T}^{\mathsf{u}}_{E_3} \uplus \mathbb{T}^{\mathsf{i}}_{E_3}$. None of these internal terms, objects and properties are visible in user code. We use the symbol @ to distinguish internal terms from user terms.

In E_3 everything, including functions, is represented as an object. In the semantics, objects are represented as records of values, where each property name is either a string m or an internal identifier @x. We denote by $\mathsf{prop}(o)$ the set of properties names defined for o. A heap H is a mapping from heap addresses l to objects o. The set of allocated heap addresses is denoted by $\mathsf{dom}(H)$.

Given a term t, we denote by $\mathsf{names}^{\mathsf{i}}(t)$ the set of top-level identifier names of t, and by $\mathsf{names}^{\mathsf{p}}(t)$ the set of all property names appearing in t (a property name p appears in t if the expression o.p appears in t). We denote by $\mathbb{P}$ the set of all identifier and property names. We partition $\mathbb{P}$ in the sets $\mathbb{P}^{\mathsf{u}}$ and $\mathbb{P}^{\mathsf{i}}$ of the user and internal property names. The identifier names present in a term always belong to the set $\mathbb{P}^{\mathsf{u}}$.

We denote by H_{E_3} the initial heap of E_3. It contains native objects for representing predefined functions, constructors and prototypes, and the global object @Global that constitutes the initial scope, and is always the root of the scope chain. We denote by $\mathcal{F}_{\mathsf{nat}}$ the set of heap addresses of all the native functions and constructors. Due to space limitations we do not list them here, but it is straightforward to obtain them from the operational semantics of JavaScript [5]. For example, in $\mathcal{F}_{\mathsf{nat}}$ there is the eval function and the Object,

Function and Array constructors. We denote by l_g the heap address of the global object @Global, and by $\mathsf{prop}_{\mathsf{nat}}(l_g)$ its set of native properties.

Program state and reduction rules. Program states in the semantics are triples H, l, t (where l is the heap address of the current scope object). The general form of a reduction rule is $\frac{<Premise>}{S_1 \rightarrow S_2}$, meaning that if a certain premise is true then the state S_1 evaluates to a state S_2. The rules can be further divided into two categories: transition axioms and context rules. An atomic transition is described by an axiom. For example, the axiom H,l,(v) $\longrightarrow$ H,l,v describes that brackets can be removed when they surround a value (as opposed to a proper expression, where brackets are still meaningful). Contextual rules propagate such atomic transitions. To satisfy the assumptions of Section III-A, we represent a state H, l, t as a pair H, t where H adds a special property @curr_scp in the global object to store the current scope l (denoted by $\mathsf{scope}(H)$).

Scope and prototype lookup. The E_3 stack is represented by a chain of objects whose properties represent the binding of local variables in the scope. Each scope object stores a pointer to its enclosing scope object in an internal @Scope property. JavaScript follows a prototype-based approach to inheritance. In the semantics, each object stores in an internal property @Prototype a pointer to its prototype object, and inherits its properties.

Property access. In E_3, each object property may be associated with the attributes readonly, dontdelete, dontenum that restrict the kind of access which can be granted on that property. Property accesses can be *explicit* or *implicit*.

Explicit property access takes place when a term explicitly names the property that is being read or written. There are three such cases: (i) the reduction of identifier x (which involves looking at objects on the scope chain, starting from the current scope object until we find an object which has the property x, either directly or in one of its prototypes); (ii) the reduction of e.x, which results in the reduction of e["x"]; (iii) the reduction of e1[e2], which involves accessing the property name corresponding to the string representation of the value obtained dynamically by evaluating the expression e2.

Implicit property access takes place when the property accessed is *not* named explicitly by the term, but is accessed as part of an intermediate reduction step in the semantics. For example, the length property is accessed with the push function of the Array.prototype object.

These accesses are important when the underlying object is the global object. We denote by $\mathcal{P}_{\mathsf{nat}}$ the set of all property names that can be accessed implicitly, defined as:

{0,1,2,...} $\bigcup$ { toString, toNumber, valueOf, length, prototype, constructor, message, arguments, Object, Array, RegExp, String, Number, Boolean, Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError}.

In E_3, the global object itself can only be accessed by using this or calling the methods valueOf of Object.prototype, or concat, sort or reverse of Array.prototype.

B. The Js subset

We now define the Js subset of E_3, so that it enforces blacklisting (and whitelisting), and it restricts code to access only those properties of the global object which appear explicitly as identifiers in the code. We shall see that Js enjoys the AuthoritySafe property, and helps in defining suitable enforcement functions to solve the isolation problem.

We begin by defining the language Ju for writing user-level programs. Next, we define Js by adding runtime checks to the Ju property access mechanisms. Both Ju and Js are parametric on a set of black-listed property names $\mathcal{B} \subseteq \mathbb{P}^{\mathsf{u}}$.

Definition 10 ($Ju^{\mathcal{B}}$): Given a blacklist $\mathcal{B}$, the subset $Ju^{\mathcal{B}}$ is defined as E_3 minus: all terms containing identifiers or property names from the set $\mathcal{B}$, all terms containing one or more of the identifiers {eval, Function, contructor}, all terms containing identifiers beginning with $.

The language $Ju^{\mathcal{B}}$ forbids access to blacklisted property names by the identifier mechanism x and the property access mechanism e.p. Given any term $t \in Ju^{\mathcal{B}}$, we define $\mathsf{enf}(t)$ as the term obtained by applying Rewrite rules 1 and 2 reported below (originally defined in [6]), which control the properties accessed via the expression e1[e2], and prevent the term this from evaluating to the global object.

Rewrite 1: Rewrite every occurrence of e1[e2] in a term by e1[IDX(e2)], where,

```
IDX(e2) =
    ($=e2,{toString:function(){return($=$String($),CHK_$)})
CHK_$ = ($BL[$] ? "bad":
                ($ == "constructor" ? "bad":
                ($ == "eval" ? "bad":
                ($ == "Function" ? "bad":
                ($[0] == "$" ? "bad":$))))))
```

where $String refers to the original String constructor, $BL is a (blacklisted) global variable containing an object with all blacklisted property names initialized to true, and $ is a reserved variable name.

Rewrite 2: Rewrite every occurrence of this by the expression (this==$g?null;this), where $g is a blacklisted global variable, initialized with the address of the global object.

Definition 11 ($Js^{\mathcal{B}}$): Given a blacklist $\mathcal{B}$, the subset $Js^{\mathcal{B}}$ is defined as $\{\mathsf{enf}(t) | t \in Ju^{\mathcal{B}}\}$.

We denote by $\mathbb{T}_{Js^{\mathcal{B}}}$ the set of all user and internal terms of $Js^{\mathcal{B}}$.

Operational Semantics of $Js^{\mathcal{B}}$. The operational semantics of $Js^{\mathcal{B}}$ is mostly the semantics of E_3 projected on the

terms in $Js^{\mathcal{B}}$, but with a different initial heap, and read-only internal properties.

Initial heap. The correctness of Rewrite rules 1 and 2 depends on the heap containing specific $-prefixed variables with the appropriate values. These are initialized by executing the following code.

Initialization Code 1: Init_BL.

```
var $String = String; var $= ""; var $g = this;
var $BL = {p1:true;...,pn:true};
```

where p1,... ,pn are the blacklisted property names.

As described earlier, a handle to the global object can be obtained by calling certain native functions, such as valueOf, sort, concat and reverse. Therefore, we wrap these functions so that the return value is never the global object. For example, in the case of valueOf we use the following initialization code.

Initialization Code 2: Init_valueOf.

```
$OPvalueOf = Object.prototype.valueOf;
$OPvalueOf.call = Function.prototype.call;
Object.prototype.valueOf =
                    function(){var $= $OPvalueOf.call(this);
                               return ($==$g?null:$)}
```

Init_sort, Init_concat and Init_reverse are defined similarly by considering the sort, concat and reverse methods respectively.

Finally, we wrap each native function $f \in \mathcal{F}_{\mathsf{nat}}$ by executing the following code.

Initialization Code 3: Init_f.

```
$f = f;
$f.call = Function.prototype.call;
f = function(){var $this = (this==$g?{ }: $this);
               return $f.call($this)}
```

We denote by Init_Fnat, the sequential composition of initialization codes Init_f for all $f \in \mathcal{F}_{\mathsf{nat}}$. We can now define the initial heap for the language Js.

Definition 12: The initial heap $H_{Js^{\mathcal{B}}}$ for the language is defined as the terminal heap obtained by executing in the heap H_{E_3} the term

Init_BL; Init_valueOf; Init_sort; Init_concat; Init_reverse; Init_Fnat.

Readonly native properties. In the initial heap $H_{Js^{\mathcal{B}}}$, we require that for all heap addresses $l \in \mathsf{dom}(H_{Js^{\mathcal{B}}}) \setminus \{l_g\}$, all the properties $p \in \mathbb{P}^{\mathsf{u}}$ of the object $H_{Js^{\mathcal{B}}}(l)$ have the *readonly* attribute set. This is a restriction imposed on the semantics of $Js^{\mathcal{B}}$ and its main purpose is to prevent a component from being able to use globally accessible locations as communication channels.

Wellformedness. We say that a heap H is wellformed for the language $Js^{\mathcal{B}}$ iff it is wellformed for the language E_3 and it can be obtained by evaluating the initial heap $H_{Js^{\mathcal{B}}}$ using a term in $\mathbb{T}_{Js^{\mathcal{B}}}$. Formally, $\mathsf{Wf}^{\mathsf{h}}_{Js^{\mathcal{B}}}(H)$ is defined as

$$\mathsf{Wf}^{\mathsf{h}}_{E_3}(H) \bigwedge \exists s, t \in \mathbb{T}_{Js^{\mathcal{B}}} : H, s \in \tau(H_{Js^{\mathcal{B}}}, t).$$

Similarly, a state H, t is wellformed iff it is wellformed according to E_3, H is wellformed according to $Js^{\mathcal{B}}$ and t is in $\mathbb{T}_{Js^{\mathcal{B}}}$. Formally, $\mathsf{Wf}_{Js^{\mathcal{B}}}$ is defined as

$$\mathsf{Wf}_{E_3}(H, t) \bigwedge \mathsf{Wf}^{\mathsf{h}}_{Js^{\mathcal{B}}}(H) \bigwedge t \in \mathbb{T}_{Js^{\mathcal{B}}}.$$

In the rest of this Section we use $\mathsf{acc}_{Js^{\mathcal{B}}}$ to denote the set of heap actions that occur during a single step transition of a wellformed state in the language $Js^{\mathcal{B}}$.

C. Authority Map for $Js^{\mathcal{B}}$

We now present an authority map for $Js^{\mathcal{B}}$ that makes the language authority safe. We begin by defining the notion of resources. A resource in $Js^{\mathcal{B}}$ is a pair of heap address and a user property name. The set of allocated resources for a given heap H are the pairs (l, p) such that l is an allocated heap address and $p \in \mathbb{P}^{\mathsf{u}}$ is any property name. Formally,

$$\mathbb{R} := \mathbb{L} \times \mathbb{P}^{\mathsf{u}} \qquad \mathsf{res}(H) := \mathsf{dom}(H) \times \mathbb{P}^{\mathsf{u}}$$

where $\mathbb{L}$ is the set of all possible heap addresses.

The set of all possible actions is $\mathbb{A} := \mathbb{R} \times \mathbb{D}$. In order to simplify notation, in the rest of this Section we will also denote actions as triplets (l, p, d) where $d \in \mathbb{D}$.

The authority of a state denotes an over-approximation of the set of all actions that it can perform on the allocated resources during its reduction. Since JavaScript is memory safe, we know that the set of heap addresses that can be accessed during the reduction of a state S are the ones that are reachable from the current scope address. Rewrite rules 1 and 2 guarantee that a term can never access a blacklisted property name. Moreover, all properties of native objects (except for the global object) are read-only, preventing write actions to be executed. First, we informally describe the authority of a state without taking into account the read-only restriction on native properties. Then, we appropriately subset the authority of each state to account for this restriction.

Definition 13 (Heap Graph): Given a heap H, we denote by $\mathcal{G}^{\mathsf{H}}(H)$ a directed graph consisting of heap addresses as nodes and an edge from node l_i to l_j labelled with $p \in \mathbb{P}$ iff $H(l_i).p = l_j$ (note that p may also be an internal property name).

Given a heap graph $\mathcal{G}^{\mathsf{H}}$, a heap address l and a set of property names $\mathcal{P} \subseteq \mathbb{P}$, we denote by $\mathsf{reach}(\mathcal{G}^{\mathsf{H}}, l, \mathcal{P})$ the set of heap addresses reachable from l by accessing property names from $\mathcal{P}$. We define the subgraph $\mathsf{proj}(\mathcal{G}^{\mathsf{H}}, l, \mathcal{P})$ as the graph defined by deleting all outward edges from l which have edge labels outside the set $\mathcal{P}$.

Without accounting for the read-only attribute on native properties, the authority map for a state H, t can be informally described as the set of all actions (l, p, d) such that one of the following holds (recall that l_g is the heap address of the global object and $\mathcal{P}_{\mathsf{nat}}$ is the set of property names that can be implicitly accessed):

1) $d = \mathsf{r}$; $l = l_g$; p is a property from the set $\mathcal{P}_{\mathsf{nat}}$;
2) $d = \mathsf{r}$; $l \neq l_g$ and l is reachable from the global object in the graph $\mathcal{G}^{\mathsf{H}}(H)$ by accessing non-blacklisted property names or property names in $\mathcal{P}_{\mathsf{nat}}$; p is a non-blacklisted user property name or a property name in $\mathcal{P}_{\mathsf{nat}}$;
3) $l = l_g$; p is an identifier name appearing in the term t or in a function stored at a heap address reachable from the current scope object in the graph $\mathcal{G}^{\mathsf{H}}(H)$;
4) $l \neq l_g$ and l is reachable from the current scope object in the graph $\mathcal{G}^{\mathsf{H}}(H)$ by accessing non-blacklisted property names or property names in $\mathcal{P}_{\mathsf{nat}}$; p is a non-blacklisted user property name or property name in $\mathcal{P}_{\mathsf{nat}}$.

To account for the read-only attribute, we subtract the actions $(\mathsf{dom}(H_{Js^{\mathcal{B}}}) \setminus \{l_g\} \times \mathbb{P}^{\mathsf{u}}) \times \{\mathsf{w}\}$ from this authority map. The formal definition of the authority map $\mathsf{auth}_{Js^{\mathcal{B}}}$ is given below. $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ correspond to points (1), (2) and (3) in the above paragraph and $\mathsf{f}_H(\emptyset, \emptyset, \mathcal{L})$ corresponds to point (4).

Definition 14 ($\mathsf{auth}_{Js^{\mathcal{B}}}$): Consider a black list $\mathcal{B}$ and a wellformed state H, t. The authority map $\mathsf{auth}_{Js^{\mathcal{B}}}(H, t)$ *can be defined as*

$$(\mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 \cup \mathsf{f}_H(\emptyset, \emptyset, \mathcal{L})) \setminus \mathcal{A}_{\not\mathsf{w}}$$

where

$\mathcal{L} = \left(\mathbb{L} \cap \{H(l_g).p \mid p \in \mathsf{names}^{\mathsf{i}}(t) \cup \mathbb{P}^{\mathsf{i}}\}\right) \cup \{l \mid l \in t\}$,
$\mathcal{A}_{\not\mathsf{w}} = \left(\mathsf{dom}(H_{Js^{\mathcal{B}}}) \setminus \{l_g\} \times \mathbb{P}^{\mathsf{u}}\right) \times \{\mathsf{w}\}$,
$\mathcal{A}_1$, $\mathcal{A}_2$ *and* $\mathcal{A}_3$ *are defined as follows*

$$\begin{array}{lcl} \mathcal{A}_1 & := & (\{l_g\} \times \mathcal{P}_{\mathsf{nat}}) \times \{\mathsf{r}\} \\ \mathcal{G}_1^{\mathsf{H}} & := & \mathsf{proj}(\mathcal{G}^{\mathsf{H}}(H), l_g, \mathcal{P}_{\mathsf{nat}}) \\ \mathcal{L}_1 & := & \mathsf{reach}(\mathcal{G}_1^{\mathsf{H}}, l_g, (\mathbb{P} \setminus \mathcal{B}) \cup \mathcal{P}_{\mathsf{nat}}) \\ \mathcal{A}_2 & := & (\mathcal{L}_1 \times \mathcal{P}_{\mathsf{nat}}) \times \{r\} \\ \mathcal{A}_3 & := & (\{l_g\} \times (\mathsf{names}^{\mathsf{i}}(t) \setminus \mathcal{B})) \times \mathbb{D}, \end{array}$$

and for sets $\mathcal{P}_1, \mathcal{P}_2$ of property names and set $\mathcal{L}$ of heap addresses, $\mathsf{f}_H(\mathcal{P}_1, \mathcal{P}_2, \mathcal{L})$ is computed by the following procedure:

1) $\mathcal{A}_i := (\{l_g\} \times (\mathcal{P}_1 \setminus \mathcal{B})) \times \mathbb{D}$
2) $\mathcal{G}_1^{\mathsf{H}} := \mathsf{proj}(\mathcal{G}^{\mathsf{H}}(H), l_g, \mathcal{P}_1)$
3) $\mathcal{L}_2 := \mathsf{reach}(\mathcal{G}_1^{\mathsf{H}}, \mathcal{L}, (\mathbb{P} \setminus \mathcal{B}) \cup \mathcal{P}_{\mathsf{nat}})$
4) $\mathcal{A}_{ii} := ((\mathcal{L}_2 \setminus \{l_g\}) \times ((\mathbb{P}^{\mathsf{u}} \setminus \mathcal{B}) \cup \mathcal{P}_{\mathsf{nat}})) \times \mathbb{D}$
5) $\mathcal{L}_3 := \{l \mid l \in \mathcal{L}_2 \wedge \mathsf{IsFunc}(H(l))\}$
6) $\mathcal{P}_3 := \{\mathsf{names}^{\mathsf{i}}(H(l).\mathit{@body}) \mid l \in \mathcal{L}_3\}$
7) *if* $\mathcal{P}_3 \subseteq \mathcal{P}_2 = \emptyset$, *return* $\mathcal{A}_i \cup \mathcal{A}_{ii}$
8) $\mathcal{L}_4 := \{l \mid \exists p \in \mathcal{P}_3 : H(l_g).p = l\}$
9) *else return* $\mathcal{A}_i \cup \mathcal{A}_{ii} \cup \mathsf{f}_H(\mathcal{P}_3 \setminus \mathcal{P}_2, \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{L}_4)$,

where for an object o, $\mathsf{IsFunc}(o)$ *is true iff o is a user-function object.*

Theorem 3: For all blacklists $\mathcal{B} \cap \mathcal{P}_{\mathsf{nat}} = \emptyset$, $\mathsf{auth}_{Js^{\mathcal{B}}}$ is a valid authority map and $\mathsf{AuthoritySafe}(\mathsf{auth}_{Js^{\mathcal{B}}})$ *holds for the language $Js^{\mathcal{B}}$.*

Proof Sketch: In order to prove this theorem, we use an invariant $\mathsf{Good}(S)$ on all wellformed states S. We prove that $\mathsf{Good}(S)$ holds for all initial states S and Good is preserved under reduction. The predicate $\mathsf{Good}(S)$ is very similar to the predicate $\mathsf{Good}_{J_{\mathsf{sub}}}(H, t)$ defined in [6]. An example of a property implied by $\mathsf{Good}(S)$ is that for all property names p present in $\mathsf{term}(S)$, $p \notin \mathcal{B}$. We split the proof of the main theorem into two parts.

Part 1: $\mathsf{auth}_{Js^{\mathcal{B}}}$ *is a valid authority map.* In order to prove this property it is sufficient to show that $\mathsf{Good}(H, t)$ and $H, t \rightarrow K, s$ imply

- $\mathsf{acc}_{Js^{\mathcal{B}}}(H, t) \subseteq \mathsf{auth}_{Js^{\mathcal{B}}}(H, t) \cup (\mathsf{act}(K) \setminus \mathsf{act}(H))$
- During the single step reduction of H, t, if a heap address l' is written to property p ($p \in \mathbb{P}$) of an object at address l, (p could be an internal property) then $\mathsf{f}_K(\emptyset, \emptyset, l') \subseteq \mathsf{auth}_{Js^{\mathcal{B}}}(H, t) \cup (\mathsf{act}(K) \setminus \mathsf{act}(H))$
- $\mathsf{auth}_{Js^{\mathcal{B}}}(K, s) \subseteq \mathsf{auth}_{Js^{\mathcal{B}}}(H, t) \cup (\mathsf{act}(K) \setminus \mathsf{act}(H))$

using the function f_H of Definition 14.
All these conditions can be proved by induction on the set of reduction rules in the operational semantics of Js. The base case follows by a cases analysis over the transition axioms and the inductive case follows from a contradiction argument on the context rules.

Part 2: $\mathsf{AuthoritySafe}(\mathsf{auth}_{Js^{\mathcal{B}}})$ *holds.* In order to prove this theorem we make use of the following property that can be derived from the operational semantics of $Js^{\mathcal{B}}$:

For all wellformed states H, t and K, v such that $H, t \rightarrow^* K, v \not\rightarrow$, for all $l \in \mathsf{dom}(H) \cap \mathsf{dom}(K)$, and $p \in \mathbb{P}^{\mathsf{i}} \setminus \{\mathit{@curr\_scp}, \mathit{@scope}\}$:

$$H(l).p = K(l).p \bigwedge K(l_g).\mathit{@curr\_scp} = l_g \bigwedge$$

$$\mathit{@scope} \in \mathsf{prop}(K(l)) \Rightarrow K(l).\mathit{@scope} = H(l).\mathit{@scope}.$$

The proof for $\mathsf{AuthoritySafe}(\mathsf{auth}_{Js^{\mathcal{B}}})$ is split in two parts. The proof that *only connectivity begets connectivity* follows by a contradiction argument. The proof that there is *no authority amplification* follows by a case analysis. □

D. Solving the Isolation Problem with Js

According to Theorem 1, in order to solve the isolation problem for a given set of forbidden actions $\mathcal{A}_{\varnothing}$, it is sufficient to define a blacklist $\mathcal{B}_{\mathsf{m}}$, an initial heap H_{m} and an enforcement function $\mathsf{enf}^k : \mathbb{T}_{Js^{\mathcal{B}_{\mathsf{m}}}} \rightarrow \mathbb{T}_{Js^{\mathcal{B}_{\mathsf{m}}}}$ for each k, such that $\mathsf{AuthorityIsolation}(\mathcal{A}_{\varnothing}, H_{\mathsf{m}}, \mathsf{enf}^1(t_1), \ldots, \mathsf{enf}^n(t_n))$ holds for any choice of terms $t_1, \ldots, t_n \in \mathbb{T}_{Js^{\mathcal{B}_{\mathsf{m}}}}$.

We choose the initial heap H_{m} to be defined as $H_{Js^{\mathcal{B}_{\mathsf{m}}}}$. For each principal id_k and term $t \in Js$, we define $\mathsf{enf}^k(t)$ as the the term obtained by replacing each identifier x in $\mathsf{names}^{\mathsf{i}}(t)$ with $id_k{::}x$, where :: denotes string concatenation. Therefore, we have $\mathsf{names}^{\mathsf{i}}(\mathsf{enf}^k(t)) = \{id_k{::}x \mid a \in \mathsf{names}^{\mathsf{i}}(t)\}$. We use this prefixing in order to separate the namespaces of the components controlled by different principals. Finally, we define $\mathcal{B}_{\mathsf{m}} = \{p \mid \exists l, d : ((l,p),d) \in \mathcal{A}_{\varnothing}\}$, where $\mathcal{A}_{\varnothing} \subseteq \mathsf{act}(H_{Js^{\mathcal{B}}})$ and moreover, for any action $a = ((l,p),d)$, we assume (i) if $a \in \mathcal{A}_{\varnothing}$ then $p \notin \mathcal{P}_{\mathsf{nat}}$ and $p \in \mathbb{P}^{\mathsf{u}}$; (ii) if $a \notin \mathcal{A}_{\varnothing}$ then also $((l, id_k{::}p), d) \notin \mathcal{A}_{\varnothing}$ for all k. Condition (i) is important because the properties in $\mathcal{P}_{\mathsf{nat}}$ are implicitly accessed and hence the user has authority over them by default.[1] Condition (ii) ensures that any of these prefixed identifier names do not clash with the forbidden identifier names. Practically, these conditions are likely to be satisfied by a natural definition of $\mathcal{B}_{\mathsf{m}}$.

Theorem 4: For the blacklist $\mathcal{B}_{\mathsf{m}}$, initial heap H_{m} and enforcement functions $\mathsf{enf}^1, \ldots, \mathsf{enf}^n$ defined above, $\mathsf{AuthorityIsolation}(\mathcal{A}_{\varnothing}, H_{\mathsf{m}}, \mathsf{enf}^1(t_1), \ldots, \mathsf{enf}^n(t_n))$ *holds.*

Proof Sketch: In order to prove the theorem, we show that

A. $\forall i, 1 \le i \le n : \mathsf{auth}_{Js^{\mathcal{B}_{\mathsf{m}}}}(H_{\mathsf{m}}, t_i) \cap \mathcal{A}_{\varnothing} = \emptyset$.
B. $\forall i, j, 1 \le i < j \le n :$ $\mathsf{auth}_{Js^{\mathcal{B}_{\mathsf{m}}}}(H_{\mathsf{m}}, \mathsf{enf}^i(t_i)) \not\triangleright \mathsf{auth}_{Js}(H_{\mathsf{m}}, \mathsf{enf}^j(t_j))$.

Condition A. We prove this condition by contradiction. Suppose there is a component t_i and an action $(l,p,d) \in \mathcal{A}_{\varnothing}$ (where d is either r or w) which is also present in $\mathsf{auth}_{Js^{\mathcal{B}_{\mathsf{m}}}}(H_{\mathsf{m}}, \mathsf{enf}^i(t_i))$. From definition of $\mathsf{auth}_{Js^{\mathcal{B}_{\mathsf{m}}}}$, it is clear that for all $(l,p,d) \in \mathsf{auth}_{Js}(H_{\mathsf{m}}, \mathsf{enf}^i(t_i))$, $p \notin \mathcal{B}_{\mathsf{m}}$ or $p \in \mathcal{P}_{\mathsf{nat}}$. By the assumptions on $\mathcal{A}_{\varnothing}$, both these cases are not possible, and we have a contradiction. Therefore, condition A must hold.

Condition B. We prove this condition by contradiction. Suppose there exist actions (l,p,w) and (l,p,r) such that $(l,p,\mathsf{w}) \in \mathsf{auth}_{Js^{\mathcal{B}_{\mathsf{m}}}}(H_{\mathsf{m}}, \mathsf{enf}^i(t_i))$ and $(l,p,\mathsf{r}) \in \mathsf{auth}_{Js^{\mathcal{B}_{\mathsf{m}}}}(H_{\mathsf{m}}, \mathsf{enf}^j(t_j))$ and $i < j$. From the definition of $\mathsf{auth}_{Js^{\mathcal{B}_{\mathsf{m}}}}$, for the heap $H_{\mathsf{m}} = H_{Js^{\mathcal{B}_{\mathsf{m}}}}$, the only object on which a term can have write authority over some property is the global object. So $l = l_g$ and $p \in \mathsf{names}^{\mathsf{i}}(\mathsf{enf}^i(t_i))$ or there exists a user function present in the heap H_{m} whose body has p as an identifier. However, since H_{m} is the initial heap the latter is not possible. Therefore we have $p \in \mathsf{names}^{\mathsf{i}}(\mathsf{enf}^i(t_i))$. Since $(l_g, p, \mathsf{r}) \in \mathsf{auth}_{Js^{\mathcal{B}_{\mathsf{m}}}}(H_{\mathsf{m}}, t_j)$, either $p \in \mathcal{P}_{\mathsf{nat}}$ or $p \in \mathsf{names}^{\mathsf{i}}(\mathsf{enf}^j(t_j))$. Since $p \in \mathsf{names}^{\mathsf{i}}(\mathsf{enf}^i(t_i))$, $p = id_i{::}p'$ for some p'. Therefore $p \notin \mathcal{P}_{\mathsf{nat}}$. This implies $p \in \mathsf{names}^{\mathsf{i}}(\mathsf{enf}^j(t_j))$ which leads to a contradiction, and we conclude that condition B must hold. □

[1]Note that all such authority is read-only and therefore cannot be used by one component to influence another (see Points 1 and 2 in the informal description of authority for $Js^{\mathcal{B}}$ in Section VI-C).

VII. Cajita is Capability Safe

The Google Caja [9] project is one of the main efforts, together with FBJS and Web Sandbox, to provide safe mashups that integrate untrusted JavaScript code. At the basis of Caja is Cajita, an object-capability subset of JavaScript. In this Section, we shall focus on Cajita, and show that as characterized here, the language is capability safe.

Google Caja. In the Caja framework, the mashup host defines a *container* page that embeds the Caja libraries and follows predefined guidelines to *tame* any other function or object that should be exposed to the untrusted mashup components. Taming is the process of making arbitrary pieces of JavaScript compatible with a safe subset, in order to offer additional mashup-specific functionalities without compromising the isolation of the untrusted components. The code of the untrusted components must belong to the *Cajita* subset of JavaScript, which is designed to be a safe object-capability language. Cajita is the core component of the Caja framework. Cajita can be used effectively to write mashup components from scratch, but it is too different from JavaScript to port existing components easily. Hence, legacy components can be written in *Valija*, which is very similar to JavaScript, and can be automatically compiled to Cajita. Compiled Valija code introduces a hefty performance cost compared to code written directly in Cajita.

A. Core-Cajita

The design principles underlying Cajita are described in several documents available online [9], but no formal specification is available. Hence, the precise details of its definition must be inferred from the publicly available implementation [9].

Besides the basic isolation principles, the Cajita implementation contains details related to several practical concerns that go beyond the scope of this paper. Here we present a concise description of what we understand to be the core Cajita language, abstracting away from features that are not essential for the isolation analysis, and simplifying many details of the enforcement mechanisms used to implement Cajita in JavaScript. Hence, our results concern the design of Cajita, rather than its current implementation. In particular, we remove from Cajita certain constructs (such as function declarations) that do not increase expressivity, but complicate the definitions. We define what constitutes the code of a valid Cajita module, making explicit some rewriting steps that would normally be hidden from the module programmer.

Definition 15 (Cajita Module): A Cajita module is a JavaScript expression of the form `function (y~){s}` *such that*

1) *s does not contain the statements* `with(e){s}`, `e in e`, `function x(x~){s}`;
2) *s does not contain the expressions* `e.p`, `e[e]`, `this`;

3) *y˜ are the free variables of s;*
4) *s cannot assign to any free variables (expressions like z = e for z∈y˜ are not allowed);*
5) *s may read and write properties of objects using the predefined functions getPub(e,e) and setPub(e,e,e), and may prevent further modification of objects using the predefined function freeze(e).*

We now explain the rationale behind this definition.

No global variables. In order for several modules to coexist without interference, and to prevent direct access to the native JavaScript objects, Cajita code is not allowed to access global variables. Instead, the container code has control over which global variables can be passed to each module. Each module is embedded in a function that takes as parameters all the free variables of the module (which for Cajita code can be statically determined). Values for such free variables must be passed by the container explicitly at runtime. For example, a module s with free variables x and y will be embedded in the function function(x,y){s}. Each attempt to write to x can be detected statically, and causes the module to be rejected.

Controlling the read/write attributes of object properties. The code of a Cajita module cannot use the expressions o.p or o["p"] to access the property p of object o. Instead, it must use the function getPub(o,p). This function implements a reference monitor that identifies the heap address of object o, the actual runtime value of property p, and based on an internal policy decides whether or not to grant read access to the property. The case of writing to object properties is analogous, and uses a function setPub. Hence, the read and write attributes of any object property are under the control of the Cajita runtime system. The only exception here is that of local variable declarations inside a Cajita module. These amount to creating and writing to properties of an activation object of a function call. These property writes, although not interposed by setPub, do not pose any threat as they remain local to the Cajita module.

Functions are not objects. In a Cajita module, functions are first-class values, but are not objects with properties: the getPub and setPub functions prevent read or write access to properties of those objects that are actually functions.

Whitelisting. Certain properties of objects, such as the Mozilla extension __proto__ can be used to compromise the enforcement mechanisms of Cajita. For that reason, by default, all properties that are not known to belong to a white-list of harmless properties are not readable or writable on Cajita objects.

Freezing objects. By default, all the whitelisted properties of a Cajita object are readable and writable. A Cajita module can use the function freeze(o) to affect the policy of the setPub function so that o becomes read-only, and can be safely shared with other modules.

B. Core-Cajita is Capability Safe

We now present a capability system for the core of Cajita, and show that the system is capability safe in accordance with Definition 9. We use the same notations and definitions used for Js.

In the capability system of Cajita, as in the analysis of Js, resources are pairs of heap addresses and property names. Formally, $\mathbb{R} := \mathbb{L} \times \mathbb{P}^{\mathsf{u}}$ and $\mathsf{res}(H) := \mathsf{dom}(H) \times \mathbb{P}^{\mathsf{u}}$. Capabilities are also pairs of heap addresses and property names. The capabilities associated with a term are the cartesian product of all the heap addresses occurring in the term with all possible property names. We associate any possible property names to each heap address in order to account for all the authority that a subject may be gain via dynamic generation of property names (in other words to prevent authority amplification). Given any capability (l, p), function desg is defined as the identity function. If l is not an activation object, the function priv is defined using the getPub and setPub functions which respectively control whether $H(l).p$ is readable and/or writable. If l is an activation object then for all property names p, $\mathsf{priv}(l, p) = \mathbb{D}$. If $H(l).p$ is not readable then getPub(l,p) returns null and if it is not writable then setPub(l,p,v) returns false. In general the value returned by getPub(l,p) and the final heap obtained after executing setPub(l,p,v) depend on the initial heap. Henceforth we make this dependency on a heap H explicit by the notation getPub_H and setPub_H. Whenever the property p is in $\mathcal{P}_{\mathsf{nat}}$ (set of implicitly accessible property names), getPub_H conservatively grants reading permissions to any object. By virtue of the semantics of Cajita, the restrictions imposed by getPub_H and setPub_H can only increase as the heap evolves during the evaluation of a term. The authority associated with a capability c is defined as the set of actions $(\mathsf{desg}(c_1), \mathsf{priv}(c_1))$ for all capabilities c_1 *reachable* from c. The set of capabilities c_1 reachable from a particular capability c is defined as the set of nodes reachable from c in the capability graph. The capability graph is a graph with capabilities as nodes, and an edge from c_1 to c_2 iff $r \in \mathsf{priv}(c_1)$ and c_2 can be obtained by reading a resource designated by c_1. Definition 17 contains a more formal description.

In full JavaScript, using the this mechanism, a native function can perform read or write operations on the global object or certain other scope objects that are passed implicitly. These actions are not accounted for in the above definition of authority. Moreover, if a term can get hold of the native eval or Function constructor, then it can use the keyword this to gain unrestricted authority over the global object. Therefore, in order for the authority map defined above to be valid, we need to impose some restrictions on the functions getPub and setPub. Let $\mathcal{F}^{\varnothing}_{\mathsf{nat}}$ denote the set of heap addresses of all native functions, such as Array.prototype.push, which can potentially involve a write action on the this object passed to them (It is

straightforward to derive $\mathcal{F}^{\varnothing}_{\mathsf{nat}}$ from the operational semantics of JavaScript). Let l_{eval} and l_{Function} denote the addresses of the eval function and Function constructor, let l_{valueOf} denote the address of the valueOf method of Object.prototype and let $l_{\mathsf{sort}}, l_{\mathsf{concat}}, l_{\mathsf{reverse}}$ denote the addresses of the sort, concat and reverse methods of Array.prototype.

Definition 16 (Restrictions): The functions setPub_H and getPub_H must satisfy the following restrictions:

1) *For all wellformed heaps H,*

$$\forall l, p : \mathit{getPub}_H(l,p) \notin \{l_{\mathit{eval}}, l_{\mathit{Function}}\} \cup \mathcal{F}^{\varnothing}_{\mathsf{nat}}.$$

2) *For all wellformed heaps H,*

$$\forall l, p : \mathit{getPub}_H(l,p) \notin \{l_{\mathit{sort}}, l_{\mathit{concat}}, l_{\mathit{reverse}}, l_{\mathit{valueOf}}\}.$$

3) *For all wellformed heaps H and for all l, p if $\mathit{getPub}_H(l,p) \neq \mathit{null}$ then $\mathit{getPub}_H(l,p) = H(l).p$.*
4) *For all wellformed heaps H and for all l, p, v if $\mathit{setPub}_H(l,p,v)$ holds then the heap state obtained just after setPub_H finishes execution satisfies- $H(l).p = v$ and $H(l').p' = K(l').p'$ for all $l \neq l'$ or $p \neq p'$.*

Capability system for Core-Cajita. We now formalize the capability system described above. The definition is parametric on the policy functions getPub_H and setPub_H.

Definition 17 (Capability System for Core-Cajita): The capability system for Core-Cajita is the tuple $(\mathbb{C}, \mathsf{desg}, \mathsf{priv}, \mathsf{tCap}, \mathsf{hCap}, \mathsf{cAuth})$*, where:*

- $\mathbb{C} := \mathbb{L} \times \mathbb{P}^{\mathsf{u}}$.
- *For all* $c \in \mathbb{C}$, $\mathsf{desg}(c) = c$.
- *For all* $(l,p) \in \mathbb{C}$*:*
  - $r \in \mathsf{priv}(l,p)$ *iff* $\mathit{getPub}_H(l,p) \neq \mathit{null}$ *or* $p \in \mathcal{P}_{\mathsf{nat}}$ *or* l *is an activation object;*
  - $w \in \mathsf{priv}(l,p)$ *iff there exists* v *such that* $\mathit{setPub}_H(l,p,v)$ *holds or* l *is an activation object;*
  - $\mathsf{priv}(c) = \emptyset$ *whenever* $\mathit{getPub}_H(l,p) = \mathit{null}$*,* $p \notin \mathcal{P}_{\mathsf{nat}}$*,* l *is not an activation object and there is no* u *such that* $\mathit{setPub}_H(l,p,u)$.
- *For all* $t \in \mathbb{T}_{\mathsf{Cajita}}$, $\mathsf{tCap}(t) = \{(l,p) \mid l \in t \wedge p \in \mathbb{P}^{\mathsf{u}}\}$.
- *For all wellformed heaps* H, $\mathsf{hCap}(H) = \{c \mid c \in \mathbb{C} \wedge \mathsf{desg}(c) \in \mathsf{res}(H)\}$.
- *For all wellformed heaps* H *and capabilities* c*,*
  - *if* $\mathsf{desg}(c) \notin \mathsf{res}(H)$ *or* $\mathsf{priv}(c) = \emptyset$ *then* $\mathsf{cAuth}(H,c) = \emptyset$*;*
  - *otherwise,* $\mathsf{cAuth}(H,c) = \{(\mathsf{desg}(c_1), d) \mid d \in \mathsf{priv}(c_1) \wedge c_1 \in \mathsf{reach}_{\mathcal{G}(H)}(c)\}$*, where* $\mathcal{G}(H)$ *is the graph with capabilities as nodes and an edge from capability* (l_1,p_1) *to capability* (l_2,p_2) *iff* $p_1 \notin \mathcal{P}_{\mathsf{nat}} \Rightarrow l_2 = \mathit{getPub}_H(l_1,p_1) \bigwedge$ $p_1 \in \mathcal{P}_{\mathsf{nat}} \Rightarrow l_2 = H(l_1).p_1$*,* *and* $\mathsf{reach}_{\mathcal{G}(H)}(c)$ *is the set of nodes reachable from* c *in the graph* $\mathcal{G}(H)$.

We are now ready to present the main result for this Section, that the above capability system is a safe capability system for Cajita. Our result only pertains to the specification of Cajita and relies on the semantic assumptions made about the functions getPub_H and setPub_H.

Theorem 5: The capability system

$$(\mathbb{C}, \mathsf{desg}, \mathsf{priv}, \mathsf{tCap}, \mathsf{hCap}, \mathsf{cAuth})$$

is a safe capability system for Core-Cajita if the functions getPub_H and setPub_H satisfy the restrictions stated in Definition 16.

Proof Sketch: Since we prove this theorem for the specification of Cajita, we assume that the functions getPub_H and setPub_H are implemented correctly and appropriately interpose all object property reads and writes as described in the definition of the language. Thus we prove the safety of the capability system with respect to a modified semantics where all object property reads and writes using the o.p or o[p] mechanism are replaced by calls to getPubH and setPubH. It is straightforward to show from the definitions that $(\mathbb{C}, \mathsf{desg}, \mathsf{priv}, \mathsf{tCap}, \mathsf{hCap}, \mathsf{cAuth})$ is a valid capability system. In order to show that the capability system is safe, we show that all the conditions in Definition 9 hold. The arguments for each of these conditions are very similar to the ones presented in the proof of Theorem 3 and are therefore omitted here. □

The above capability system for Core-Cajita gives us a natural authority map for which the AuthoritySafe property holds (Theorem 2). From Theorem 1 we know that, in order to solve the isolation problem for a Cajita mashup built with components $t_1, \ldots, t_n$ and a set of forbidden actions $\mathcal{A}_{\varnothing}$, we must appropriately define the initial heap H_m and enforcement functions $\mathsf{enf}^1, \ldots, \mathsf{enf}^n$ such that $\mathsf{AuthorityIsolation}(\mathcal{A}_{\varnothing}, H_m, \mathsf{enf}^1(t_1), \ldots, \mathsf{enf}^n(t_n))$ holds. One solution is to define define enforcement functions which "bind" capabilities with non-influencing authority to distinct components. For example, if getPub_H and setPub_H are forced to be disjoint for all H, then any two capabilities designating different resources will always have non-influencing authority. This can be achieved by imposing that for all l, v, $\mathsf{setPub}_H(l,p,v)$ returns false for any $p \in \mathcal{P}_{\mathsf{nat}}$. If $p \notin \mathcal{P}_{\mathsf{nat}}$, then either $\mathsf{getPub}_H(l,p)$ holds or $\mathsf{setPub}_H(l,p,v)$ holds but not both. Summarizing, authority isolation can be achieved by simply binding capabilities designating disjoint sets of resources to distinct components, thereby providing a solution to the isolation problem.

C. Comparison with Js

In Section IV, we defined the sub-language Js of E_3 and showed that it is authority safe. The language Js is similar in spirit to Cajita, and achieves authority safety by

controlling the identifiers present in a term. However, *Js* is more expressive than Cajita, and allows the this and with construct. This additional expressivity comes at the cost that a term can access global variables. As a result, the authority associated with a term is strongly dependent on the heap, and it is difficult to define any non-trivial notion of capabilities such that the capabilities held by the term are heap-independent. This conflicts with the capabilities paradigm where the capabilities held by a term should be heap-independent.

VIII. Related Work

Capability-based protection is a widely known method for operating-system-level protection, deployed in such systems as the Cambridge CAP Computer, the Hydra System, StarOS, IBM System/38, and the Intel iAPX423, all summarized in [10]. Another interesting operating system project is Amoeba [11], in which servers respond to messages sent with capabilities. Among other interesting features, the Amoeba system had a concept of owner capability, which had all rights, and the ability of someone holding an owner capability to compute a capability with more restricted rights (using exclusive-or and a cryptographic hash function).

Among prior operating system research, the closest to the present paper (to the best of our knowledge) is an analysis and proof of the EROS confinement mechanism [24], which uses an operational semantics of system execution. While there are some other similarities between their framework and our general setup, one substantial difference is that instead of defining authority as an over-approximation of heap actions that can be performed by a single object, they define authority for the whole system. Therefore, the main theorem shows that if a system performs an action, the system has authority to perform that action; this does not give precise information about whether a part of the system that performed the action was allowed to do so. Of course, the study of EROS involves system call actions, not the programming language actions studied in this paper.

Previous work on the object-capability model (e.g., [12], [13], [19]) outlines a number of principles and discusses advantages of the model. However, principles and goals are presented using the reference graph, which is intended to represent the references held by each object and define the authority of an object as the aggregated authority of all reachable objects. While reference graphs are a pictorial and intuitive way to explaining how authority can be transferred from one object to another, we have not located technical work, similar to the present paper, that connects the object-capability model to operational semantics (or other semantics) of programming languages that are intended to conform to the model. In particular, we do not know of any semantically-based proofs of confinement for object-capability languages such as E, Joe-E, Emily, W7 [14], [15], [16], [17], [20].

Some recent work by Murray, Lowe and others (e.g., [25]) defines precise forms of object-capability models in the context of process calculus.

While language-based research on information flow (see [26]) has similar goals to the present work – protecting systems from untrusted coded that operates as part of the system – there are a number of significant differences. In particular, information flow properties are normally properties of specific programs, based on program analysis, while the present paper is concerned with whether and entire language or sub-language supports authority confinement principles.

IX. Conclusions

Because important modern web sites such as OpenSocial [1] platforms, iGoogle [2], Facebook [3], and Yahoo!'s Application Platform [4] allow third parties to contribute JavaScript applications that are not trusted by the sites or their other users, there is a widely recognized need to provide isolation. While past research has focussed on protecting trusted code embedded in the hosting page from untrusted applications, we show that current systems do not effectively isolate separate third-party applications from each other. In fact, one of the attacks we discovered while analyzing isolation between FBJS applications also defeats the entire sandbox and allows an untrusted application to redefine the way that functions are applied to arguments.

We focus on capability-based protection and reachability properties of the object-capability model [12], [13]. Although a number of languages have been previously designed as object-capability languages, including E [14], Joe-E [15], Emily [16], and W7 [17], no previous study of these languages has related the object-capability model to the semantics of programs in a way that would support rigorous proofs of properties of actual code. We identify a subset of the object-capability goals with reachability consequences that we call authority safety and give precise definitions of both authority safety and a form of capability safety, based on programming language semantics. We prove that (i) capability safety implies authority safety, and (ii) authority safety is sufficient to provide isolation.

We study two language examples in some detail. In Section VI, we identify a subset *Js* of JavaScript and show that it is authority safe. This implies that *Js* supports isolation between untrusted applications. However, *Js* does not appear to support the object-capability model and our proof methods do not involve capabilities. In Section VII, we prove capability safety for the core language of Cajita, an object-capability subset of JavaScript used in Caja. By our general theorems connecting capability safety to authority safety and isolation, this implies isolation between untrusted applications that are provided capabilities with non-influencing authority. While Caja uses a surface language Valija that is translated to Cajita, we leave analysis of full Caja and the Valija to Cajita translation to future work. We

also hope to extend the analysis of *Js* and Cajita to the recently standardized ECMAScript 5 semantics and consider more general forms of mashups.

Additional directions for future research include further study of the object-capability model and analysis of previously proposed object-capability languages such as E [14], Joe-E [15], Emily [16], and W7 [17]. Interesting aspects of the object capability model that have not been previously formalized using programming language semantics include encapsulation, defensive consistency and the principle of least authority. We believe that further analysis of these concepts will provide additional insight into the problem of building secure mashups with complex interacting components.

Acknowledgments. We thank Mark Miller and the Google Caja team for invaluable comments and discussions. We are indebted to Andrei Sabelfeld and anonymous reviewers for their comments and suggestions. Maffeis is supported by EPSRC grant EP/E044956/1 and Mitchell and Taly acknowledge the support of the National Science Foundation and the Office of Naval Research.

REFERENCES

[1] O. Foundation, "OpenSocial," http://www.opensocial.org/.

[2] Google, "iGoogle," http://www.google.com/ig.

[3] The Facebook Team, "Facebook," http://www.facebook.com/.

[4] Yahoo! Inc., "Yahoo! Application Platform," http://developer.yahoo.com/yap/.

[5] S. Maffeis, J. Mitchell, and A. Taly, "An operational semantics for JavaScript," in *Proc. of APLAS'08*, ser. LNCS, vol. 5356. Springer Verlag, 2008, pp. 307–325.

[6] ——, "Isolating JavaScript with filters, rewriting, and wrappers," in *Proc. of ESORICS'09*. Springer Verlag, 2009, See also: Dep. of Computing, Imperial College London, Technical Report DTR09-6.

[7] D. Crockford, "ADsafe: Making JavaScript safe for advertising," http://www.adsafe.org/, 2008.

[8] The Facebook Team, "FBJS," http://wiki.developers.facebook.com/index.php/FBJS.

[9] G. Caja Team, "Google-Caja: A source-to-source translator for securing JavaScript-based web," http://code.google.com/p/google-caja/.

[10] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.

[11] A. Tanenbaum, R. V. Renesse, H. V. Staveren, G. Sharp, S. Mullender, J. Jansen, and G. V. Rossum, "Experiences with the amoeba distributed operating system," *Communications of the ACM*, vol. 33, pp. 46–63, 1990.

[12] M. Miller, K.-P. Yee, and J. Shapiro, "Capability myths demolished," Johns Hopkins University, Tech. Rep., 2003.

[13] M. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, Johns Hopkins University, 2006.

[14] E. Rights, "The E language," http://erights.org/elang/index.html.

[15] A. Mettler and D. Wagner, "The joe-e language specification (draft)," U.C. Berkeley, Tech. Rep. UCB/EECS-2006-26, May 2006.

[16] M. Stiegler, "Emily: A high performance language for enabling secure cooperation," in *Proc. of C5 '07*, 2007, pp. 163–169.

[17] J. A. Rees, "A security kernel based on the lambda-calculus," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 1996.

[18] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Caja: Safe active content in sanitized JavaScript," hhtp://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf.

[19] F. Spiessens, "Patterns of safe collaboration," Ph.D. dissertation, Université Catholique de Louvain, 2007.

[20] F. Spiessens and P. V. Roy, "The Oz-E Project: Design guidelines for a secure multiparadigm programming language," in *Multiparadigm Programming in Mozart/OZ*, ser. LNCS, vol. 3389. Springer Verlag, 2005, pp. 21–40.

[21] S. Maffeis, J. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," Dep. of Computing, Imperial College London, Technical Report DTR10-04, 2010.

[22] S. Maffeis and A. Taly, "Language-based isolation of untrusted Javascript," in *Proc. of CSF'09*. IEEE, 2009.

[23] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer, "Talking to strangers without taking their candy: isolating proxied content," in *SocialNets '08*. ACM, 2008.

[24] J. Shapiro and S. Weber, "Verifying the eros confinement mechanism," in *Proc. of IEEE S&P'00*. Washington, DC, USA: IEEE Computer Society, 2000, p. 166.

[25] T. Murray and G. Lowe, "Analysing the information flow properties of object-capability patterns," Proc. of FAST'09, 2009.

[26] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE J. Selected Areas in Communications*, vol. 21, 2003.